THÈSE

Préparée au
Laboratoire d'Analyse et d'Architecture des Systèmes du CNRS

Présentée pour obtenir le titre de
Docteur de l'Université de Toulouse,
délivré par l'Institut National Polytechnique de Toulouse

École doctorale Systèmes, spécialité Systèmes Informatiques

Par M. Ludovic Courtès

Sauvegarde coopérative de données pour dispositifs mobiles

Cooperative Data Backup for Mobile Devices

Soutenue le 23 novembre 2007 devant le jury composé de :

M. David Powell	Directeur de thèse
M. Marc-Olivier Killijian	Co-directeur de thèse
M. Hans-Peter Schwefel	Rapporteur
M. Pierre Sens	Rapporteur
M. Michel Banâtre	Membre
M. Claude Castelluccia	Membre
M. Ivan Frain	Membre
M. Yves Roudier	Membre

Preface

This dissertation presents my PhD thesis work made at the *Laboratoire d'Analyse et d'Architecture des Systèmes* of the French National Center for Scientific Research (LAAS-CNRS), a research laboratory located in Toulouse, France. This work was done in the Dependable and Fault-Tolerant Computing research group (TSF), under the direction of David Powell and Marc-Olivier Killijian.

The research topics tackled in this dissertation stem from the MoSAIC project[1] (*Mobile System Availability, Integrity and Confidentiality*), a three-year project funded by the French national program for Security and Informatics. This work was also partly financed by the Hidenets European project (*Highly Dependable IP-Based Networks and Services*) and the ReSIST European Network of Excellence (*Resilience for Survivability in IST*).

I am thankful to all the people who contributed to this thesis through their cooperation, help and support, be it as colleagues, friends, or family members. An extended summary in French of this dissertation is available and includes acknowledgements.

[1] *http://www.laas.fr/mosaic/*

Contents

List of Figures

Introduction

Mobile computing devices such as laptops, personal digital assistants (PDAs) and mobile phones are increasingly relied on but are used in contexts that put them at risk of physical damage, loss or theft. Their owners use them to actually create new data. Indeed, digital cameras as well as video and sound recorders may be used to create very large amounts of data. Moreover, as mobile computing devices become smaller, more powerful, and more versatile, they are being used in more and more areas of our daily lives, and are increasingly depended on.

However, few mechanisms are available to improve the *dependability* of mobile devices and of data stored on them. Available mechanisms such as data "synchronization" tools suffer from shortcomings, such as requiring manual intervention from the user, and their use is often constraining (e.g., users must be in the physical vicinity of their desktop computer). This essentially leaves users of mobile devices with at best intermittent opportunity for data backup.

We believe this situation calls for improved data backup mechanisms making the best use of the capabilities of current mobile devices and their usage patterns. Mobile devices are becoming *ubiquitous* and able to communicate with neighboring devices. With the advent of *ad hoc* network technologies, opportunistic networking is becoming a reality, allowing devices in vicinity of each other to spontaneously form networks, without any human intervention. These observations prompted us to explore the possibilities of leveraging spontaneous interactions among mobile devices with the aim of providing a *cooperative data backup service.*

Cooperative services are not an entirely new idea. "Peer-to-peer" services have been increasingly deployed on the Internet, showing that large-scale resource sharing can provide considerable synergy. Services have been deployed that can manage amounts of resources and handle quantities of requests that would be very difficult to achieve with a centralized approach. Furthermore, such decentralized cooperative services display interesting dependability properties: they have no single point of failure and no single point of trust. Naturally, our proposal of a cooperative backup service for mobile devices has been influenced by these successes.

This dissertation is organized as follows. Chapter 1 provides background information and gives a brief overview of the topics covered in the dissertation. Subsequent chapters address specific aspects of the design, evaluation and implementation of a cooperative backup service for mobile devices. At the end of each chapter, a section summarizes related work, followed by a summary of the contributions of the chapter. Key words of ideas discussed in their vicinity have been placed in the outer margin and are provided as an aid to document browsing.

Chapter 2 describes our motivations, describes our dependability goals and provides a high-level view of the cooperative backup service we propose. Chapter 3 is concerned with the analytical evaluation of the dependability of the service. The next chapter, Chapter 4, focuses on the design of suitable distributed data storage mechanisms and provides an experimental evaluation of some mechanisms. Chapter 5 addresses some of the security concerns that arise in situations requiring cooperation among distrustful principals. Finally, Chapter 6 puts all the pieces together, showing how our results were integrated into a practical cooperative backup implementation. We then conclude on the contributions of this dissertation and identify future research tracks.

Chapter 1. Approach and Contributions

This chapter provides background information about dependability and ubiquitous computing, along with a preview of topics developed in the rest of the dissertation, with forward references to chapters where each is explained. At the end of those chapters are summaries of related work.

1.1. Background

This section provides background about the two main topics addressed in this thesis, namely *dependability* and *ubiquitous computing.*

1.1.1. Dependability and Fault-Tolerance

The work presented in this thesis aims to make mobile computing systems more *dependable.* To that end, it proposes the design of specific *fault-tolerance mechanisms* for mobile devices. As such, our work falls into the computer fault-tolerance domain, a sub-domain of computer dependability.

Both domains are active research areas, with their own established concepts and terminology [Avizienis *et al.* 2004]. Dependability is usually defined as the ability to deliver a service that can be justifiably trusted. A number of *threats* to dependability have been identified and are summarized here:

- A *service failure* is an event that occurs when the delivered service deviates from the correct service, i.e., it is a transition from correct service to incorrect service provision.
- An *error* is the part of the total state of the system that may lead to its subsequent service failure.
- Finally, a *fault* is the adjudged or hypothesized cause of an error.

This yields a causal chain where faults may lead to errors, which may in turn lead to failures. Fault that actually yield an error are *active* while others are *dormant.* In composed systems, the failure of a subsystem can be seen as a fault of the higher-level system that uses it. In computing systems, faults can be, for instance, hardware faults (e.g., a design or implementation defect of a processor) or software faults (e.g., a programming error). A *vulnerability* can be thought of as an internal fault that enables an external fault to cause a system error.

Dependability is characterized by a number of *attributes* listed below [Avizienis *et al.* 2004]:

- *availability* characterizes the readiness for correct service;
- *reliability* characterizes the continuity of correct service;
- *safety* characterizes the absence of catastrophic consequences on the user(s) and the environment;
- *integrity* characterizes the absence of improper system alterations;
- *maintainability* characterizes the ability to undergo modifications and repairs;
- *confidentiality* characterizes the absence of unauthorized information disclosure; it is referred to specifically when addressing *security* concerns of dependability.

Security is characterized by a subset of these attributes, namely confidentiality, integrity and availability.

To make systems dependable, thereby guaranteeing (some of) these attributes, a number of *means* have been commonly used. The authors of [Avizienis *et al.* 2004] list the following:

- *fault removal* aims to reduce the number and severity of faults;
- *fault prevention* aims to prevent the occurrence or introduction of faults;
- fault tolerance *fault tolerance* aims to avoid service failures in the presence of faults;
- *fault forecasting* aims to estimate the present number, the future incidence, and the likely consequences of faults.

These approaches are complementary: while fault removal and prevention are concerned with *fault avoidance*, fault tolerance and forecasting relate to *fault acceptance*. For instance, fault removal may not be trusted to remove all faults; therefore, fault tolerance can be used to complement it by tolerating residual faults. Likewise, fault-tolerance mechanisms may only tolerate a limited range of faults, hence the need for fault removal.

In this thesis, we focus on improving the *availability* of the critical data stored on mobile devices. We achieve this goal by designing and implementing *fault-tolerance mechanisms* for mobile devices that replicate their data opportunistically, thereby leveraging cooperation with other mobile devices.

1.1.2. Ubiquitous Computing and Mobile *Ad Hoc* Networks

Over the last decade, electronic devices have become smaller while still becoming more powerful. Today's mobile electronic devices have processing power comparable to that of not-so-old desktop computers. Most mobiles phones and portable media players, for instance, are capable of decoding and rendering compressed video streams in real-time; personal digital assistants (PDAs) and "smart phones" can run the same office production suite that is commonly used on desktop computers. Both miniaturization and improved capabilities have allowed for new applications and have slowly led mobile computing devices to be, indeed, *pervasive.*

The emergence of short- to medium-range wireless communication means has further improved the usability of mobile devices. It is now possible, in a glimpse, to enable a mobile phone to communicate with a desktop computer (e.g., *via* a Bluetooth or Zygbee wireless link—a process also known as *device pairing*), or to get a laptop computer to use the available *wireless local area network* (WLAN), while requiring little or no manual intervention from the user. Various technologies allow for the creation of such wireless networks among mobile devices in an *ad hoc* manner, spontaneously, without requiring the installation of any networking infrastructure. The resulting networks may be maintained *cooperatively* by the set of devices that participate in it. These are usually referred to as *mobile ad hoc networks* (MANETs).

mobile ad hoc networks

While currently not widely deployed, MANETs are envisioned as a means for communication in contexts where reliance on a network infrastructure or deployment thereof is not feasible, such as emergency situations and deployment of sensor networks. They also allow the development of new cooperative services that either replace or bridge the gap left by infrastructure-based approaches. These include metropolitan area mesh networks (MANs) as well as inexpensive Internet access provision in rural or poor regions [Jain & Agrawal 2003]. Examples include the *Roofnet* urban mesh network [Bicket *et al.* 2005] and the "One Laptop Per Child" (OLPC) educational project. OLPC is planning to equip its laptops with mesh networking technologies as a means to improve (Internet) connectivity in developing regions of the world [One Laptop Per Child Project 2007]. Because they are inherently decentralized, such technologies empower users and promote cooperation.

Enabling technologies include the IEEE 802.11 standard [IEEE-SA Standards Board 1999], also known as *Wi-Fi.* Communication in MANETs is limited to devices within radio range, unless a *packet routing protocol* is used to allow remote devices to be reached in several *hops*, using neighboring devices as *relays.* Popular routing algorithms for MANETs include AODV [Perkins *et al.* 2003] and OLSR [Clausen & Jacquet 2003]; the forthcoming IEEE 802.11s standard aims to achieve wider acceptance of such technologies. As devices move, the set of reachable devices changes and the quality of radio links varies. Thus, MANETs are characterized by *intermittent and unpredictable connectivity.*

Wi-Fi

packet routing protocol

Additionally, the storage capabilities of mobile devices have increased rapidly, again allowing for new applications. Devices that can fit in your pocket may carry gigabytes of data. Many mobile devices are also capable of actually producing significant amounts of data, notably capture devices such as digital cameras, sound recorders, etc. Usually, the data stored on mobile devices either comes from or is to be stored at a desktop computer: portable audio/video players usually play the role of cache for data available "at home", while data produced on PDAs, digital cameras or sound recorders are usually eventually stored on a personal desktop computer or the Internet.

This thesis focuses on new fault-tolerance mechanisms called for by mobile devices. Mobile devices are increasingly relied on but are used in contexts that put them at risk of physical damage, loss or theft. However, fault-tolerance mechanisms available for these devices often suffer from shortcomings. For instance, data "synchronization" mechanisms, which allow one to replicate a mobile device's data on a desktop machine, usually require that the desktop machine be either physically accessible or reachable *via* the Internet. Use of third-party backup servers typically also requires access to some network infrastructure. Therefore, our goal is to devise fault-tolerance mechanisms for mobile devices that leverage their characteristics and build upon mobile networks and cooperation among devices.

1.2. Thesis Statement

In response to our observations, we set out to demonstrate the following thesis statement:

> It is feasible to improve the dependability of mobile computing devices by leveraging opportunistic cooperation among such devices.

Concretely, we propose a *cooperative backup service* that opportunistically replicates a mobile device's critical data at neighboring devices, using *ad hoc* wireless communication means.

1.3. Overview

This section briefly presents the various research domains related to the work presented in this thesis. Namely, we focus on fault-tolerance mechanisms for mobile devices, then describe related work in distributed data storage, notably in a mobile context, and finally present related work on secure cooperation among mutually suspicious principals.

1.3.1. Fault-Tolerance Mechanisms for Mobile Devices

Although mobile devices are at risk of loss, theft, or physical damage, little work has been dedicated to fault-tolerance for mobile devices, and in particular data backup. Allowing mobile devices to tolerate failures, be they transient (e.g., accidental deletion of a file) or permanent (e.g., loss or theft), requires being able to replicate the device's data. From the user's viewpoint, this essentially consists of the data produced or modified using the device: documents, email archives, pictures, sound recordings, etc.

In practice, data available on mobile devices is usually replicated using data "synchronization" mechanisms, which allow one to replicate a mobile device's data on a desktop machine, using an appropriate synchronization protocol. Such a mechanism usually requires that the desktop machine be either physically accessible or reachable *via* the Internet. Another option, use of third-party backup servers, typically also requires access to some network infrastructure. While some infrastructure-based networking technologies (e.g., GSM/GPRS, UMTS) have wide geographical coverage, using them may be costly. This severely reduces the availability of these mechanisms since a network infrastructure may not always be reachable as one moves around. Consequently, current approaches only moderately reduce the risk of data loss.

Mechanisms have been proposed to improve on this situation while avoiding reliance on an infrastructure. The basic idea is similar to that of peer-to-peer resource sharing networks, such as those widely used on the Internet for information sharing (e.g., sharing music, videos or software) or computational power sharing: mobile devices contribute storage resources for others to use and in return opportunistically replicate their data at neighboring devices. This approach has been taken for backup purposes within a personal area network (PAN) with devices all belonging to the same user [Loo *et al.* 2003]. It has also been acknowledged as a way to facilitate data sharing among mobile devices as we will see later. In this thesis, we take a similar cooperative approach and apply it to backup among mobile devices. Our goal is to allow anyone to participate in the cooperative backup service. Device owners do not necessarily have *a priori* trust relationships, so devices are mutually *distrustful*.

delay tolerant networks

From a networking perspective, research on *delay tolerant networks* (DTNs), which primarily aims to cope with very high latency networks and intermittent connectivity, has led to solutions comparable to the cooperative backup approach [Zhang 2006]: if a mobile device sends backups to a desktop computer via a DTN, the backups travel through a number of mobile devices acting as *relays* before they eventually reach the desktop computer. Our approach differs from the use of DTNs in that it is specifically designed with data backup in mind and addresses security issues not addressed by DTNs.

replication strategies

erasure codes

Our dependability goals will be described in Chapter 2, along with a design overview of the envisioned cooperative backup service. Chapter 3 will provide an analytical evaluation of dependability gains provided by the service. Several *replication strategies* are considered, including the use of *erasure codes*.

1.3.2. Distributed Data Storage

The cooperative backup service we propose relies on the storage of data items (backups) on neighboring mobile devices. Therefore, it shares a number of concerns with distributed data stores, and in particular mobile distributed stores. On the Internet, large distributed data stores are used for file sharing. As nodes participating in a peer-to-peer store come and go unpredictably, such systems need to support data fragmentation and scattering, and need to be able to maximize data availability despite node turnover. Since participating nodes are mutually suspicious, peer-to-peer distributed stores are further concerned with data integrity: it must be possible to detect data items that have been tampered with by an attacker. The design of our cooperative backup service shares these requirements.

Not surprisingly, Internet-based peer-to-peer cooperative backup systems are also a valuable source of inspiration. They provide solutions tailored to the storage and transmission of backup data, and some of them are also designed to cope with untrusted contributors.

The mobile context is even more constraining than Internet peer-to-peer systems since mobile devices may become unreachable for longer periods and as the number of nodes reachable may vary significantly. A further challenge is that mobile devices are usually resource-constrained: they have limited bandwidth, storage and processing power, and more importantly have limited energy. All these have been driving issues in the design of the storage mechanisms for our cooperative backup service.

The cooperative backup service presented in this thesis builds on techniques used in both Internet-based and mobile distributed stores. Specifically, Chapter 4 will describe the requirements placed on the storage layer of our cooperative backup software, as well as solutions to address them. These include the provision of data integrity and confidentiality guarantees, handling of data fragmentation and scattering through the design of a suitable meta-data schema, as well as compression methods. Section 4.4 will provide an experimental evaluation of the storage layer in terms of storage efficiency and processing power requirements. Our major contribution in the distributed storage area lies in the choice and adaptation of storage techniques to our target environment.

1.3.3. Secure Cooperation

We aim to leverage the ubiquity of communicating mobile devices to provide opportunistic data replication. Such an approach works best if any mobile device can participate in the service, without requiring prior trust relationships among participants. Internet-based peer-to-peer services paved the way for such open, cooperative services. They showed that openness allowed rapid large-scale deployment. Therefore, we opted for such an open approach.

Openness comes at the cost of the risk of non-cooperative behavior by some participants. An attacker could participate in the service with the goal of selfishly draining all (storage) resources, or could try to exhaust all the resources dedicated to the service in its vicinity. Likewise, a malicious participant could lie and purposefully fail to provide the backup service. These and other *security threats* can severely impede cooperation among

security threats

mobile devices or even lead to denial of service (DoS), unless provisions are made to address them.

More generally, since participants have no prior trust relationships, their cooperation decisions must take into account these risks and make arrangements to reduce them. While some arrangements made at the storage layer partly address them, in particular by enforcing data confidentiality and integrity and by providing redundancy, additional mechanisms need to be introduced at the cooperation level.

In Chapter 5, we propose a decentralized approach to these security issues that allows a certain level of *accounting.* We argue that accounting is a building block that enables cooperation among distrustful principals. Our solution leaves users with the freedom to choose from a wide range of *cooperation policies.* We discuss the impact of Sybil attacks, a kind of attack that is inherent to systems that do not rely on a centralized naming authority. An actual implementation of the proposed security primitives is also sketched.

1.3.4. Implementation and Evaluation

Finally, Chapter 6 puts all the pieces together: it outlines the design and implementation of our cooperative backup software. In particular, it shows how the storage facilities presented in Chapter 4 are used, and how they are complemented by higher-level storage facilities that handle file meta-data, collections of files and versioning.

Our implementation of the cooperative backup service consists of a *backup daemon* and a set of *clients* allowing the end-user to interact with it. Most "backup activities" are implemented by the backup daemon, namely: input data indexing and opportunistic replication, data retrieval, and storage provision. These activities are mainly triggered by several well-defined events. The algorithm used by each of them is given in this chapter.

We also describe a series of experimental measurements that allowed us to obtain preliminary results concerning the performance of our implementation. We conclude on lessons learnt from the current implementation.

Chapter 2. Design of a Cooperative Backup Service for Mobile Devices

Mobile devices are increasingly relied on but existing data backup mechanisms suffer from shortcomings that effectively put data stored on mobile devices at risk. This chapter presents the rationale and dependability goals that led us to design a cooperative backup service for mobile devices. It then outlines the envisioned cooperative backup service and discusses related work.

2.1. Rationale

This section describes the problem we are trying to solve and outlines the cooperative backup approach we took.

2.1.1. Problem Statement

Mobile computing devices such as laptops, personal digital assistants (PDAs) and mobile phones are increasingly relied on but are used in contexts that put them at risk of physical damage, loss or theft. As the processing power and storage capacities of these devices increase, new applications arise that allow them to be used in new ways. Their owners use them to actually create new data, as they would do with their desktop computers. Capture devices such as digital cameras as well as video and sound recorders, may be used to create large amounts of data. As mobile computing devices become smaller, more powerful, and more versatile, they get used in more and more areas of our daily lives, and are increasingly depended on.

However, fault-tolerance and in particular data backup mechanisms available for these devices often suffer from shortcomings. The most common way to create backups of the data available on a mobile device is through data "synchronization" mechanisms such as SyncML [Open Mobile Alliance 2001]. Most mobile devices implement such a protocol, allowing users to synchronize the data on their device with that stored on their desktop machine. This procedure is usually bi-directional: data created or modified on the mobile device is sent to the desktop machine and *vice-versa*. Such synchronization mechanisms require that the desktop machine be either physically accessible or reachable *via* the

Internet. Another option consists in using third-party servers such as those provided by `box.net`, which also requires access to a network infrastructure.

Unfortunately, in many scenarios where devices are carried along in different places, access to a network infrastructure (e.g., *via* a Wi-Fi access point) is at best *intermittent*. Often, access to a network infrastructure may be too costly and/or inefficient energy-wise and performance-wise to be considered viable "just" for backup. In emergency situations and upon disaster recovery, for instance, infrastructure may well be unavailable for an unknown amount of time. In such cases, data produced on a mobile device while the network is unreachable cannot be replicated using the aforementioned synchronization techniques and could be lost. Similarly, sparsely populated rural regions [Jain & Agrawal 2003] and environments with scarce Internet connectivity, such as those targeted by the "One Laptop per Child" project (OLPC, *http://laptop.org/*), can hardly rely on access to an infrastructure for doing data backup.

As a result of these intermittent connectivity patterns, backup opportunities may occur infrequently. Consequently, data produced on mobile devices is at risk. Our goal is to improve the dependability of data stored on mobile devices with only intermittent connection to an infrastructure.

2.1.2. A Cooperative Backup Approach

The service we envision, which we call MoSAIC[1], aims to improve the dependability of the data stored by mobile devices by providing them with mechanisms to tolerate hardware or software faults, including permanent faults such as loss, theft, or physical damage. To tolerate permanent faults, our service must provide mechanisms to store the user's data on alternate storage devices using the available communication means.

Mobile devices equipped with wireless communication means are becoming ubiquitous. Their storage capacity keeps growing, as is their processing power. At the same time, *impromptu* and *ad hoc* communication technologies have emerged. Considering this, we believe that a *cooperative* backup service could leverage the resources available in a device's neighborhood. Such a service is operated by the users and for the users, in a decentralized fashion. Devices are free to participate in the service. They can benefit from it by storing backups of their data on other devices. They are expected to contribute to it in return by donating storage and energy resources for the service. In the sequel, we use the term *contributor* when referring to a device acting in its role of storage provider; we use the term *data owner* when referring to a device in its role of "client", i.e., one that uses storage provided by the contributors to replicate its data. For fairness, participating devices are expected to act as both data owners and contributors.

contributor

data owner

[1] *Mobile System Availability, Integrity and Confidentiality*, *http://www.laas.fr/mosaic/*. MoSAIC is partly financed by the French national program for Security and Informatics (ACI S&I). Our partners are IRISA (Rennes) and Eurécom (Sophia-Antipolis).

Commonly used Internet-based peer-to-peer services have paved the way for cooperative approaches, particularly for cooperative storage. While relying on the infrastructure provided by the Internet, they have shown how special-purpose *overlay networks* could be built on top of it. The computer nodes participating in such networks cooperate to achieve a specific task such as the maintenance of a large distributed store [Dabek *et al.* 2001, Kubiatowicz *et al.* 2000, Goldberg & Yianilos 1998, You *et al.* 2005] or that of a large anonymous communication network [Kügler 2003, Clarke *et al.* 2001]. For fault-tolerance and data backup purposes, the cooperative backup approach on the Internet has gained momentum over the last few years. Cooperative archival and backup systems have been proposed [Lillibridge *et al.* 2003, Cox *et al.* 2002, Cox & Noble 2003, Landers *et al.* 2004, Goldberg & Yianilos 1998]. An important fault-tolerance benefit is that cooperative backup services may be deployed on a wide range of *heterogeneous* hardware and software platforms, thereby reducing the probability of a catastrophic failure. The usability of cooperative services is also very good since launching the software is all it takes to start using them [Cox *et al.* 2002]. The synergy resulting from such large-scale resource sharing makes Internet-based cooperative services both very valuable and cheap: one just needs to dedicate a small amount of storage resources to use a very large distributed store.

overlay networks

It is our belief that similar advantages can be offered by a cooperative data backup service in the mobile context. There are key differences with Internet-based approaches, though. First, the maintenance of an overlay network atop some network infrastructure appears unnecessary in an infrastructure-less context. One of the primary goals of overlay networks on the Internet is to facilitate node discovery and allow for spontaneous interactions among them. *Ad hoc* networking plays this role in the mobile context: mobile devices can *spontaneously* interact as they encounter each other physically. Second, mobile device connectivity may be much looser than that of Internet peer-to-peer network nodes. In particular, network partitioning is much more likely among a set of mobile nodes connected *via ad hoc* links than among nodes connected through the Internet, notably because communicating mobile nodes have to be in the same *physical* vicinity. Third, due to partner mobility, fixed pair-wise relations among devices cannot be maintained. Instead, devices have to adapt and cooperate with new devices. Fourth, as a consequence of increased network partitioning, resource sharing occurs at a much smaller scale in a mobile context. These specificities raise a number of additional challenges.

A cooperative backup service for mobile devices allows one to benefit from backup mechanisms even in the absence of an infrastructure. In addition, even when an infrastructure is technically reachable (e.g., GPRS, UMTS, IEEE 802.16 aka. WiMAX, or even satellite phones), it provides a cheaper alternative, both financially and, to some extent, in terms of energy requirements. From a practical viewpoint, using short-range technologies such as Wi-Fi saves the relatively high cost of GPRS/UMTS transmission fees and subscriptions. It also provides the benefits of using a wide range of heterogeneous storage devices, that may belong to different administrative domains: no contributing device is a *single point of failure* and none is a *single point of trust*. In other words, the cooperative backup service will

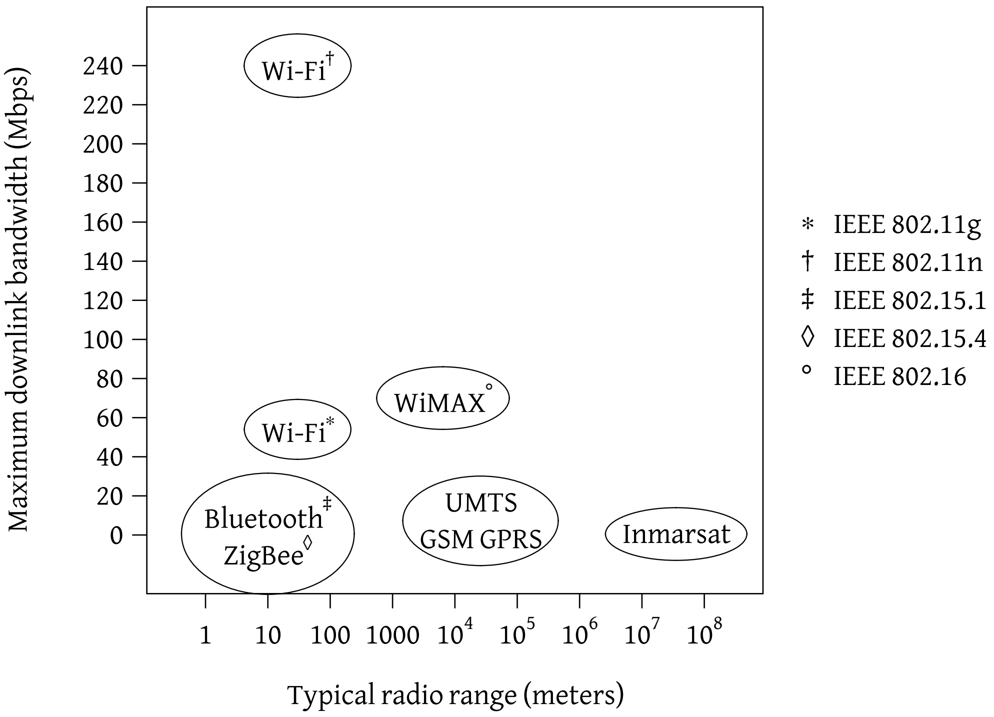


Figure 1. Comparison of the downlink bandwidth and typical radio range of several wireless data transfer technologies.

take advantage of ubiquitous storage devices in such a way that failure or compromise of any single contributing device cannot result in a failure of the backup service.

Comparing the energy consumption of infrastructure-based and infrastructure-less technologies appears to be complex, though. The maximum output power of wireless interfaces is roughly proportional to their coverage range: GSM/GPRS and UMTS terminals, for instance, have a maximum output power of up to 1 W, compared to 100 mW for Wi-Fi and as little as 1 mW for ZigBee. On the other hand, infrastructure-based technologies can make use of protocols that significantly reduce the energy requirements of terminals by transferring part of the burden to base stations, which infrastructure-less communication techniques cannot do. For instance, both 802.11 [IEEE-SA Standards Board 1999] and 802.15.4 [Zheng & Lee 2004], when used in infrastructure mode or in indirect data transmission mode, respectively, provide energy-saving modes where the base station or coordinator can store messages on behalf of mobile devices until they leave their energy-saving mode and ask for pending messages. Conversely, when used in *ad hoc* mode, Wi-Fi consumes a large amount of energy, even if the device does not actually receive or transmit data [Feeney & Nilsson 2001].

Nevertheless, while energy consumption of current *ad hoc* wireless technologies may still be an issue, forthcoming technologies hold the promise of reduced power consumption. In the area of personal area networks, ZigBee offers good energy efficiency characteristics, yet at the cost of lower bit rates [Zheng & Lee 2004]. Machines of the OLPC project aim to be energy-efficient by having the 802.11s-style mesh routing protocol implemented at the hardware level so that OLPC computers can act as routers for the mesh network even when powered off [One Laptop Per Child Project 2007]. Similarly, GSM and UMTS cell phones are becoming increasingly energy-efficient. For instance, music streaming services are being offered for UMTS phones[2]; some mobile phone operators even provide video streaming services for UTMS phones. The mere existence of such bandwidth-intensive mobile applications illustrates a reduction of the energy requirements of wireless networking technologies. Research carried out at the application level also appears promising. For example, proposals have been made to turn radio interfaces off based on predicted traffic (or rather, predicted lack of traffic); experiments have shown that such a pragmatic approach can indeed noticeably reduce power consumption [Zhang *et al.* 2005].

Another advantage of our approach is that short-range wireless communication means can provide higher bandwidth, particularly uplink bandwidth, than long-range, infrastructure-based protocols. Figure 1 shows the maximum downlink bandwidth and typical radio range of several current networking technologies; the uplink bandwidth is usually slightly lower than the downlink bandwidth. Practically, this allows larger amounts of data to be backed up when a backup opportunity occurs.

On the other hand, the outcome of our cooperative backup service will be highly dependent on the frequency of participant encounters, a parameter that we can hardly influence. Nevertheless, as long as some participant encounters occur, there is an opportunity for dependability improvement. Among other things, the analytical evaluation presented in Chapter 3 aims to characterize scenarios under which our approach is beneficial compared to making backups only when an infrastructure is available.

2.2. Dependability Goals

As already stated, our primary goal is to improve the dependability of data stored on mobile devices, especially compared to current approaches to data backup for these devices. In particular, we want to protect devices against permanent data loss resulting from, e.g., physical damage, loss or theft. We aim to do so by leveraging cooperation among a set of devices with no prior trust relationship. From a dependability and security viewpoint, this approach comes at a cost: since contributing devices are not trusted, no strong guarantees can be made about service provision. This raises an important challenge: providing

[2] *Vodafone and Sony to Launch Music Service*, Gizmodo, January 2006, *http://gizmodo.com/gadgets/cellphones/vodafone-and-sony-to-launch-music-service-147402.php.*

mechanisms that make the backup service itself *dependable* despite the lack of trust in contributors. We summarize threats arising from the cooperation with untrusted devices and depict the mechanisms we envision to tackle them.

2.2.1. Threats to Confidentiality and Privacy

There is an obvious threat to confidentiality when it comes to storing critical data on untrusted devices: a malicious storage contributor may try to access data stored on behalf of other devices. Therefore, confidentiality has to be provided at the storage layer and is achieved through end-to-end data encryption, as will be discussed in Chapter 4. Thanks to this, an eavesdropper listening to storage requests over the network cannot gain any more information about the contents of the data being backed up than the contributor itself. Thus, communication eavesdropping is not a serious additional threat to confidentiality and the communication layer does not need to provide any additional encryption.

Privacy of the participating users can also be threatened. An eavesdropper may be able to know *whether* a device is actively transferring data, and it may be able to estimate the amount of data being replicated. It may also be able to know the parties involved (the physical devices or even their owner), especially when in their physical vicinity. Recent attempts have been made to support anonymity in MANETs, for instance based on anonymous multi-hop routing [Rahman *et al.* 2006] or multicast trees [Aad *et al.* 2006]. However, we do not address threats to privacy in detail in this dissertation. Nonetheless, we hope to provide a minimum level of identity privacy by allowing users to use self-managed identifying material (which may not establish any binding with their real-world identity, i.e., *pseudonyms*), rather than compelling the use of identifying material provided by a central authority.

2.2.2. Threats to Integrity and Authenticity

There are also evident threats to data integrity and authenticity: a malicious contributor could tamper with data stored on behalf of other nodes, or it could inject garbage data that would pass all the integrity checks performed by data owners but would not be of any use to the data owner.

Integrity threats also arise at the communication layer: an intruder may try to tamper with messages exchanged between two devices, thereby damaging the data being backed up. Thus, the communication layer must also guarantee the integrity of messages exchanged between participating devices.

2.2.3. Threats to Availability

Unavailability threats against the cooperative backup service fall into two categories: unavailability resulting from accidental data loss (including accidental failure of contributors holding replicas), and data or service unavailability resulting from *denial of service* (DoS) attacks committed by malicious nodes.

denial of service

Obviously, data unavailability due to accidental failures of either the owner or contributor devices is the primary concern when building a cooperative backup service.

Malicious participating devices may also try to harm individual users or the service as a whole, denying use of the service by other devices. A straightforward DoS attack is *data retention*: a contributor either refuses to send data items back to their owner when requested or simply claims to store them without actually doing so. DoS attacks targeting the system as a whole include *flooding* (i.e., purposefully exhausting storage resources) and *selfishness* (i.e., using the service while refusing to contribute). These are well-known attacks in Internet-based peer-to-peer backup and file sharing systems [Lillibridge *et al.* 2003, Cox *et al.* 2002, Bennett *et al.* 2002] and are also partly addressed in the framework of *ad hoc* routing in mobile networks [Michiardi & Molva 2002, Buttyán & Hubaux 2003]. These threats can be seen as *threats to cooperation.*

2.2.4. Discussion

Security threats related to the data being backed up, in particular threats to data availability, confidentiality, and integrity are largely addressed by the storage layer of our cooperative backup service. Chapter 3 describes and evaluates replication strategies aiming to improve data availability. Chapter 4 details our cooperative backup storage layer and how it addresses data confidentiality and integrity issues.

Given the risks of non-cooperative behavior, we believe that cooperation can only be leveraged if the cooperative service supports *accountability.* In our view, accountability is a building block upon which users can implement their own higher-level *cooperation policies* defining the set of rules that dictate how they will cooperate. Chapter 5 proposes core mechanisms to provide accountability and discusses cooperation policies that may be implemented on top of it.

accountability

2.3. Backup and Recovery Processes

This section details the backup and recovery processes we envision for our cooperative backup service. Techniques to achieve our dependability goals and to address the aforementioned threats are outlined.

2.3.1. Backup Process

First, we expect users to synchronize or replicate the data available on their mobile devices on their desktop machine while in its vicinity. While this is not a strict requirement, we expect it to be a likely use case that needs to be taken into account. The cooperative backup software can take advantage of this by only replicating newly created or modified data not already available on the desktop machine.

When moving around with their mobile devices, users should leave them on. As already mentioned, while current wireless networking technologies are a serious energy drain [Feeney & Nilsson 2001], we believe that chances are that these will become more energy-efficient in the not-so-distant future. The cooperative backup software running on the mobile device is autonomous. It should automatically discover network infrastructures reachable using its wireless networking interface, or, in the lack of an infrastructure (i.e., in *ad hoc* mode), it should discover neighboring devices that contribute to the cooperative backup service. The former is typically achieved using operating system facilities. On the other hand, discovery of service providers in the *ad hoc* domain can be achieved using an appropriate *service discovery protocol* (SDP).

service discovery protocol

Service discovery protocols have been widely deployed on fixed local area networks (LANs) for some time. They allow the discovery of neighboring printers, workstations, or other devices, while requiring little or no configuration, making users lives easier. Popular service discovery protocols for LANs include SMB (*Server Message Block*, mainly used by Microsoft operating systems), DNS-SD (*Domain Name System Service Discovery*, initially introduced in Apple's Mac OS X operating system where it is known as *Bonjour* [Cheshire & Krochmal 2006a]), SLP (*Service Location Protocol*, [Guttman *et al.* 1999]) and SSDP (*Simple Service Discovery Protocol* [Goland *et al.* 1999], part of UPnP). This wealth of protocols specifically targets LANs. Because of their network usage characteristics, they do not adapt well to MANETs, and in particular to resource constraints (energy and bandwidth). Because of this, research has gone into the design of SDPs suitable for MANETs [Sailhan & Issarny 2005, The UbiSec Project 2005, Helmy 2004, Poettering 2007, Kim *et al.* 2005]. To our knowledge, no particular SDP for MANETs has been widely deployed to date. We do not focus on the design of such SDPs. Instead, we will use one of these protocols for our purposes.

Once a data owner wishing to replicate its data has discovered contributing devices, it chooses one of them and sends it a backup request. The choice of a contributing device can be made based on a number of criteria. Most likely, these criteria will reflect trust and cooperation concerns: "Is this contributor a friend of mine?" "Have I already interacted with this contributor before?" "Did this contributor previously provide me with a valuable backup service?" "Is this contributor selfish?" Similarly, the contributor can then either accept or reject the request, again depending on a number of criteria such as trust, cooperation fairness, and local resource availability. For instance, it may choose to reject requests from strangers, unless it has a large amount of free space and energy.

Storage requests will typically consist of small data blocks rather than, e.g., whole files. This is needed since connections between two contributors are unpredictable and may be short-lived. In most cases, since the data owner does not trust the contributor, the data will be sent encrypted along with meta-data allowing for integrity checks upon recovery. Data owners may maintain a local database indicating which contributors have offered them storage resources. This information can then be used as an input to *cooperation decisions*. Similarly, when sending a data block, data owners may wish to embed the contributor's identifier within the block, so that the contributor can eventually be marked as "more trustworthy", should the block become recoverable.

For most scenarios, it would be unrealistic to rely only on chance encounters between devices for recovery. Thus, eventually, when they gain access to the Internet, contributors should send the data stored on behalf of data owners to an *agreed upon on-line store*. This store needs to be reliable and trusted by data owners. In practice, it could be implemented in various ways: it could be a large store shared among all participants (e.g., a centralized FTP server, a distributed peer-to-peer store, etc.), or a per-owner store (e.g., the data owner's own email box, etc.) This dissertation does not focus on the implementation of this on-line store.

Note that contributing mobile devices could as well *forward* data blocks or further *replicate* them on behalf of other nodes. However, we chose to not rely on these possibilities for several reasons. First, if all mobile devices obtain access to the Internet as frequently as each other, there is nothing to be gained from data forwarding: data blocks will not reach the on-line store any faster. Without additional information on the frequency of contributors' Internet access, data block forwarding is not appealing. Additional issues with data forwarding stem from the lack of trust in contributors.

A store-and-replicate strategy where contributors replicate (as opposed to merely forwarding) data blocks by themselves is more appealing. However, this approach is comparable to *flooding* in multi-hop networks [Zhang 2006]. Thus, a potential issue is resource exhaustion, should this strategy be followed by a large proportion of contributors: since contributors can hardly coordinate themselves to make reasonable use of the available resources, they could end up saturating neighboring storage resources. Routing protocols address this issue by limiting the scope of flooding, for instance by annotating packages with a time-to-live (TTL) indicating when a packet can be dropped. Nevertheless, we believe flooding remains too resource-intensive for our purposes. More importantly, it would make the replication process "uncontrolled", whereas data owners may prefer to have tight control over replication of their data rather than blindly entrusting contributors to operate well. In particular, because a store-and-replicate strategy would be more costly (energy-wise) to contributors than simple data storage, they can hardly be trusted to implement it. These issues will be further discussed in Chapter 3.

2.3.2. Recovery Process

In a purely *ad hoc* scenario where data owners never gain access to the infrastructure, data owners could query neighboring devices to recover their data. However, as already mentioned, we consider that most scenarios would rather use a specific on-line store as an intermediary between contributors and data owners. Upon recovery, data owners query that on-line store for their data. To that end, they may need to provide the on-line store with identifying material proving that they are the legitimate owner of the data. Most of the time, they will ask for the latest snapshot available, but they could as well ask for, say, "the snapshot dated April 4th".

Data owners then recursively fetch, decipher, decode and check the integrity of all the data blocks that made it to the on-line store. If blocks contain information about the contributor that stored them, they may update their contributor trust database (or some cooperation incentive service), increasing the trustworthiness of those devices that contributed to this backup.

Since all contributors gain Internet access at their own pace, it may be the case that data owners need to wait before all data blocks reach the on-line store. Furthermore, if a contributor created and replicated several snapshots while in the *ad hoc* domain, all the blocks comprising each of these snapshots may eventually reach the on-line store. Thus, when recovering data from the on-line store, a data owner might be able to recover a more recent version, provided they wait long enough. However, without additional information (e.g., when recovering from a permanent failure), data owners cannot known *whether* more recent versions will eventually be available in the on-line store. This shows a tradeoff between data *freshness* and data *availability*.

2.4. Related Work

This section describes related work in the areas of fault-tolerance mechanisms and state replication, peer-to-peer cooperative backup, mobile and distributed storage, as well as delay-tolerant networking.

2.4.1. Checkpointing and Rollback

In this thesis, we focus on fault-tolerance for individual mobile devices when viewed as centralized systems whose service provision (e.g., taking pictures, editing documents, etc.) does not necessarily depend on other devices, as opposed to distributed systems involving distributed computations among a set of nodes[3]. We summarize the main system-level and application-level approaches. These approaches differ regarding whether snapshots are made upon user requests (i.e., "user-directed") or upon system request (e.g., periodically), and regarding the data manipulated (i.e., "user data" such as files, or whole program data). Figure 2 summarizes the different approaches and their characteristics.

2.4.1.1. System-Level Approaches

checkpointing

Fault-tolerance of centralized systems is often achieved by logging changes to their state, by taking snapshots of their state at regular intervals (also referred to as *checkpointing*),

[3] For the distributed case, a good survey on checkpointing techniques can be found in [Elnozahy 2002].

	user-directed snapshots	**system-directed snasphots**
user data, files	version control systems (VCS)	file system backup software
	database management systems (DBMS)	
	versioning file systems	
program data	“object prevalence”	persistent language runtimes
		persistent OSes
	application checkpointing	

Figure 2. Different ways of handling checkpointing.

or by a combination of both. Upon recovery, the last state before the system failed can be re-installed by either replaying the changes listed in the system’s log, or by reinstating the last snapshot. To tolerate permanent faults of the system, the log or snapshot must be stored on one or several different devices.

For the checkpointing operation itself to be dependable, it must honor several properties. Namely, checkpointing must be *atomic*: either it occurs or it does not; it must not leave a mixture of the former and new snapshots of the system. Snapshots must be *consistent*: the available snapshot must reflect correct system state, both before and after a new snapshot has been made. These properties are part of the *ACID properties* (Atomicity, Consistency, Isolation, Durability) used to qualify *transactions* in database management systems (DBMS) [Gray 1981]. Another way to say this is that checkpointing must have *transactional semantics*.

ACID properties

In practice, several approaches have been proposed to provide system checkpointing. A global approach is that of *persistent operating systems*. In a persistent OS, all the data structures used by the OS kernel and the processes running on top of it are regularly written to disk [Liedtke 1993, Shapiro & Adams 2002]. The OS takes care of taking snapshots and ensures that this is an atomic operation. This approach is often referred to as *orthogonal persistence* because both the user and application developer are freed from concerns related to checkpointing. However, persistent OSes described in the literature do not handle state replication. Thus, they can only tolerate transient faults (e.g., temporary power outage). In addition, they keep only one checkpoint, which does not allow recovery of previous states. A similar approach, although limited to single applications, is taken by checkpointing libraries such as [Plank *et al.* 1995]. On Figure 2, such libraries are represented as “application checkpointing”; since applications can also instruct the library to make a checkpoint, this approach appears in both columns.

orthogonal persistence

Other approaches provide users and/or application programmers with direct control of *when* a snapshot should be made. These allow them to tell the system when a transaction begins and when it ends. Snapshots can then only be made in between transactions.

DBMSes support such user-directed transactions. Most implementations provide easy-to-use database replication mechanisms. Likewise, proposals were made to integrate transactional storage with programming languages [Gray 1981, Birrell *et al.* 1987]. This approach is being applied to a wide range of programming languages such as Java [Prevayler.Org 2006], Common Lisp and Ruby, and is now often referred to as *object prevalence*. This term comes from the fact that application objects can be checkpointed without requiring the programmer to explicitly convert them to some external presentation. So-called *object-relational mapping* (ORM) is a similar approach, aiming to fill the gap between objects implemented by the programming language and a corresponding persistent representation stored in a DBMS; a widely used ORM implementation is Hibernate, for Java [Red Hat, Inc. 2007].

2.4.1.2. Application-Level Checkpointing

Persistent OSes and run-times are not widely deployed today. Thus, in practice, persistence and checkpointing are usually implemented at the application level. However, in the majority of cases, *ad hoc* mechanisms are used, where only part of the application data are actually made persistent; in addition, the ACID properties are rarely honored by applications, unless they rely on programming language support or DBMS. Indeed, most applications use the OS-provided *file system* for long-term storage.

POSIX file systems, do not support transactional semantics; instead, they provide applications with an interface that only allows for in-place updates, an approach that was long acknowledged as detrimental to fault-tolerance [Gray 1981]. Thus, on top of commodity OSes such as Unix derivatives and Windows, data that has to be made persistent is simply written (in a non-atomic way) to the file system. This is rarely referred to as "checkpointing" given that only part of the application's state is actually saved, and given that it is up to the application to implement its own state capture mechanisms. In effect, commodity OSes force programmers to deal with two distinct memory models: memory is globally consistent and has an all-or-nothing durability model, whereas file systems usually offer no global consistency model and file content durability depends on various implementation details [Shapiro & Adams 2002].

Most of the time, additional replicas of user data stored in the file system are created asynchronously, using a backup application that periodically walks the file system and sends the file to one or several storage devices or backup servers. This approach is implemented by the traditional Unix tool `dump` and by similar tools such as Burt [Melski 1999], Rsnapshot [Rubel 2005], or those found in the Plan 9 operating system [Quinlan & Dorward 2002]. Again, the state that is replicated in this way may be inconsistent, for instance if applications were writing to a file while the backup was being made.

versioning

To provide rollback while guaranteeing checkpoint atomicity, an approach that has been proposed is the use of *immutable objects* and *versioning*: instead of updating existing objects by mutating them, new objects are created [Gray 1981][4]. Version control systems, such as GNU RCS [Tichy 1985], GNU Arch [Lord 2005], and Git [Hamano 2006], as well as versioning file systems like WAFL [Hitz *et al.* 1994], Elephant [Santry *et al.* 1999], Ext3COW [Peterson & Burns 2003] and Wayback [Cornell *et al.* 2004], provide users with transactional operations in addition to the POSIX interface: users can modify their data and issue a `commit` request to record a transaction, i.e., to store a new snapshot. This is illustrated on Figure 2 where versioning files systems appear in both columns.

This approach is particularly useful for software development and has been used as a basis for *software configuration management* (SCM) systems: since transactions are committed by developers, each transaction can be made to contain a logical, consistent set of changes. Furthermore, snapshots can be annotated, which allows developers to, e.g., roll-back to a previous "known-good" state of the project. Finally, dependencies of a software project can be expressed using *time-domain addressing* [Gray 1981], for instance by specifying the exact version of each dependency or build tool that is required; since each version is an immutable snapshot, this makes software builds reproducible. In particular, Compaq's SCM system, Vesta [Heydon *et al.* 2001], and the Nix package management system [Dolstra & Hemel 2007] are designed around these ideas.

2.4.2. Peer-to-Peer Cooperative Backup

This section summarizes related work on cooperative backup, both in the Internet and mobile *ad hoc* domains.

2.4.2.1. Internet-based Cooperative Backup

Data backup has traditionally been implemented using centralized techniques. Typical backup software [Melski 1999, Rubel 2005] periodically sends data to a single or a small set of previously specified servers. For the backup servers to not be a *single point of failure*, the collected data are then usually copied to fault-tolerant storage devices, such as RAID (Redundant Array of Inexpensive Disks). To tolerate disasters, such as a fire in the server room, the data are then frequently copied to removable storage devices that are then stored in a different place. Many companies implement a backup scheme along these lines.

However, such backup schemes require a costly and complex infrastructure, which individuals or even small companies cannot necessarily afford. Several companies sell storage resources for backup purposes on their Internet-accessible storage servers but again, these offers may be limited (in terms of bandwidth and available space), still relatively costly and are centrally managed. To address this issue, Internet-based *cooperative* backup systems that build on the peer-to-peer resource sharing paradigm have been proposed and implemented. While a cooperative approach can potentially leverage large amounts of unused resources, as was shown with the advent of peer-to-peer file sharing, they also

[4] Interestingly, functional programming languages usually follow this paradigm by avoiding or completely impeding object mutation.

benefit from the inherent diversity of participating machines. Since different and physically separated machines are unlikely to fail at the same time, diversity can improve fault-tolerance.

With the Cooperative Internet Backup Scheme [Elnikety *et al.* 2002, Lillibridge *et al.* 2003], interested users subscribe to a central server. They can then query the central server for a list of candidate *partners*, i.e., machines that would also like to contribute resources to participate in the service. Once partners have been selected, a negotiation occurs and partners agree on some amount of storage resources and machine uptime. Pair-wise relations are established among partners. Each partner can then periodically send its backup data to the other partners; each partner must also accept storage requests from its partners and honor restoration requests.

Another cooperative system, *Pastiche*, uses a similar approach [Cox *et al.* 2002]. In Pastiche, each node also has a set of partners with which it trades storage resources. Storage resource trades in Pastiche are symmetrical: each participant owes its contributors as much resources as it was given. Partner discovery, however, is completely *decentralized*: Pastiche relies on a peer-to-peer *overlay network* where participants can look for partners according to various criteria. One such criterion may be the *degree of data overlap* between the requester and the candidate peer: this allows partner selection to be biased towards peers already storing similar data, thereby reducing the amount of data that needs to be transmitted among partners.

Other cooperative backup systems such as *pStore* [Batten *et al.* 2001] and *Venti-DHash* [Sit *et al.* 2003] use a decentralized storage model. Instead of having each participant maintain pair-wise relations with a set of peers, the data of all participants is stored in a global peer-to-peer distributed store. The distributed store is implemented on top of *Chord*, a data location service based on a *distributed hash table* (DHT). Briefly stated, a distributed hash table is a distributed data structure akin to regular hash tables that maps identifiers (e.g., data item identifiers) to nodes (e.g., IP addresses of machines storing the designated data items). Distributed stores built on DHTs account for the intermittent connectivity of peers by pro-actively replicating data items, so that they can remain available despite the failure of a small fraction of nodes.

distributed hash table

However, maintaining the content stored in a DHT can be quite bandwidth-intensive, particularly when participating nodes come and go at a high rate. Since the amount of data stored in such a large-scale backup system can be quite important, this can quickly become an issue. To address this, *PeerStore* [Landers *et al.* 2004] uses a hybrid approach. Each node in PeerStore has a set of "preferred" partners, with which it is directly connected. In addition, nodes may consult a global-scale DHT before replicating data: this allows them to detect whether the data are already stored by nodes outside their set of partners, in which case they can avoid replicating them again.

From a security perspective, all these systems are meant to be used by nodes with no *a priori* trust relationships. Thus, they all provide protection against data confidentiality and integrity attacks, as discussed in Section 2.2. However, Pastiche and pStore, for instance,

	Peer Discovery	**Data Storage**	**Data Versioning**
Cooperative Internet Backup Scheme	Centralized	Controlled (pair-wise)	No
Pastiche	Decentralized	Controlled (pair-wise)	Yes
pStore	Decentralized	Decentralized	Yes
PeerStore	Decentralized	Decentralized	Yes

Figure 3. Comparison of Internet-based cooperative backup systems.

both assume that participants are well-behaved and follow the protocol. Conversely, other systems such as Samsara and the Cooperative Internet Backup Scheme provide better protection against DoS attacks such as those presented earlier. Another cooperative backup system, BAR-B [Aiyer *et al.* 2005], focuses primarily on a addressing these threats to cooperation based on a formal reasoning. These security aspects will be explored in Chapter 4 and Chapter 5, respectively.

Except for the Cooperative Internet Backup Scheme, all these systems use an immutable storage model, where previously stored data can be erased (at best) but not modified; in addition, they allow several versions of the user's file system to be backed up. Upon restoration, the user can choose a specific version. Other Internet-based systems have been described in the literature or implemented, but they share most of the characteristics of those mentioned above [Cooley *et al.* 2004, Peacock 2006, Stefansson & Thodis 2006, Martinian 2004, Morcos *et al.* 2006, Allmydata, Inc. 2007]. Figure 3 summarizes the approaches to peer discovery and data storage taken by these cooperative backup systems.

Although not specifically designed with data backup in mind, large-scale distributed stores described in the literature are worth mentioning. *OceanStore* [Kubiatowicz *et al.* 2000] aims to provide a large-scale *persistent* store that is maintained cooperatively by a set of untrusted servers. Multiple versions of each data item may also be kept. Likewise, *InterMemory* [Goldberg & Yianilos 1998] aims to provide long-term archival storage thanks to a large distributed store maintained by a large number of untrusted peers on the Internet. They defer from the aforementioned peer-to-peer backup systems in that the archived data is expected to be of public interest (e.g., digitalized content from a library, scientific data repositories) or at least shared by a group of people (e.g., calendar, email) and may therefore be accessed by a large number of nodes. Consequently, both systems opt for a large distributed store, rather than for pairwise relations as found in, e.g., Pastiche. Thus a large part of the work consists in providing efficient data location and routing algorithms, access control, as well as automated data replication as contributing nodes come and leave the global store, all of which are not primary concerns of peer-to-peer backup systems.

Distributed data stores that target file sharing, such as CFS [Dabek *et al.* 2001], do not provide data persistence guarantees. File sharing systems whose goal is to provide anonymous publishing and retrieval, such as GNUnet [Bennett *et al.* 2003] and Freenet [Clarke *et al.* 2001], do not guarantee data persistence either. This makes them unsuitable for data backup and archival purposes.

2.4.2.2. Cooperative Backup for Mobile Devices

personal area network

To our knowledge, few attempts have been made to adapt the cooperative backup paradigm to the mobile environment. *Flashback* provides cooperative backup for mobile devices within a *personal area network* (PAN) [Loo *et al.* 2003]. Similarly, *OmniStore*, although primarily aiming at seamless replication among one's own portable devices within a PAN, allows newly created data to eventually be replicated to an on-line "repository" such as a stationary computer [Karypidis & Lalis 2006]. This approach differs from ours in that the devices involved in the cooperative backup service all belong to the same person. Thus, they are all mutually trusted[5], so some of the security threats we described in Section 2.2 do not apply. In particular, data confidentiality and integrity attacks, as well as DoS attacks are unlikely in this context.

Another important difference is that devices within the PAN are expected to be within radio range of each other most of the time. In particular, the recovery process in Flashback can only take place when the devices involved in the backup can reach each other, assuming that any personal device that has become unreachable will *eventually* be reachable again. Tight connectivity among one's personal devices allows for *lazy replication* (i.e., replicate when deemed appropriate), as opposed to the *opportunistic replication* (i.e., replicate anytime a backup opportunity occurs) that we envision in MoSAIC.

Flashback and OmniStore also introduce interesting ideas as to how to *discover* devices and select the most "useful" devices by taking into account the power and storage resources of all the personal devices, as well as their pair-wise *locality factor* (i.e., how often a device is reachable by another one) [Loo *et al.* 2003, Karypidis & Lalis 2005]. These pieces of information are passed to a device utility function which is used during the device selection process. The choice of a utility function that accounts for both device power budget and locality shows that it increases the lifetime of devices by favoring the use of the most capable devices while avoiding the weakest ones [Loo *et al.* 2003]. Using co-location information could be applied to our cooperative backup service. However, using information about a device's power budget would not be feasible in our case since devices are mutually suspicious and consequently could lie about their power budget, or could refuse to disclose it.

[5] This assumes that devices are not stolen, in which case an attacker may be able to exploit the cooperative backup mechanism to retrieve information stored by other devices (e.g., by reprogramming the stolen device and reinserting it into the PAN as a Trojan).

A master thesis studied scenarios similar to those we are interested in, involving cooperative backup among distrustful mobile devices [Aspelund 2005]. Its approach differs from ours in that it focuses on scenarios where data recovery is performed in *ad hoc* mode by contacting contributors through a multi-hop routing scheme. The major contribution of this work lies in the assessment through simulation of the distance of data owners to their data backup in terms of number of hops in the *ad hoc* domain. It does not propose actual storage mechanisms and protocols, and does not address security concerns such as those described earlier.

Finally, although in a different application context, distributed persistent data storage for wireless sensor networks (WSNs) addresses concerns close to those of cooperative backup systems for PANs: data collected by the sensors is to be replicated among a set of trusted devices, typically with good connectivity [Girao *et al.* 2007].

2.4.3. Mobile and Distributed Storage

Over the last decade, a large body of research has been dedicated to distributed data storage for mobile devices. Most of the proposed storage systems use data replication among mobile devices to maximize data availability despite intermittent connectivity. Many also cope with frequent disconnection due to node mobility using cooperative storage schemes. Thus, although those systems are not concerned with data backup *per se*, their solutions are relevant in the context of the design of a cooperative backup service.

Existing literature on the topic can be categorized according to several criteria. First, essentially two usage patterns are considered: *information sharing* following a write-once read-many model (WORM), and *read-write data access* allowing data items to be modified in-place by mobile nodes and eventually propagated. The former is an adaptation of peer-to-peer file sharing systems to the mobile context, while the latter is an extension of (distributed) file systems to the mobile environment.

2.4.3.1. Mobile Information Sharing

Peer-to-peer file sharing systems for mobile devices are very similar to Internet-based file sharing systems. A distributed store is maintained by a set of cooperative mobile nodes, each contributing storage resources. No central service is relied on and participating peers all play the same role. Users can share data by inserting it in the cooperative store. Usually, the inserted data is immutable: it cannot be updated in-place, and can hardly be completely removed on demand (it may eventually vanish from the store, though, for instance if it is not popular enough, or if network partitioning occurs). Immutability usually allows a wide range of replication and caching techniques to be used by contributing nodes.

Such a system was proposed by Sözer *et al.* [Sözer *et al.* 2004]. It adapts techniques used by wire-line peer-to-peer protocols by, e.g., taking nodes physical locality into account to limit communication to nodes within radio range. Similarly, ORION [Klemm *et al.* 2003] and the Hybrid Chord Protocol [Zöls *et al.* 2005] focus on adapting existing peer-to-peer protocols to the mobile context to reduce the traffic load and energy consumption yielded by the replication and data lookup protocols.

Several read-only *cooperative caching* strategies have been proposed. 7DS implements cooperative caching of (read-only) data available on the Internet among mobile devices to increase the data availability of participants not connected to the Internet [Papadopouli & Schulzrinne 2000]. Participating devices cache information as they access it through the Internet and then spread cached information on-demand in the *ad hoc* domain. Tolia *et al.* described the design of a read-only cooperative cache for distant file servers that works similarly [Tolia *et al.* 2004]. Wang *et al.* proposed a cooperative caching framework aiming to maximize data locality [Wang *et al.* 2006]. A similar cooperative caching strategy has been proposed in [Flinn *et al.* 2003]; however, it differs from the other approaches in that it was explicitly designed to deal with *untrusted* contributors.

2.4.3.2. *Writable Distributed Storage*

Mutable distributed storage for mobile devices has been envisioned as a means to *extend* users' stationary file systems to their mobile devices. Essentially, one's mobile devices act as a *cache* of data stored on a stationary device that is viewed as a *server*. The ability to modify data creates *cache consistency issues* not relevant to read-only file sharing systems. Namely, modifications made on a device need to be propagated to other devices. In the case where different changes are made by several devices not connected to each other, these changes must eventually be *reconciled* when they get connected to each other, so that a single new version of the data is agreed on. Because of mobility and intermittent connectivity, only *weak consistency* can be maintained among replicas [Demers *et al.* 1994].

cache consistency issues

With *OmniStore* [Karypidis & Lalis 2006], data from the user's desktop computer is transparently replicated to (some of) their mobile devices; mobile devices within the user's personal area network (PAN) can then lazily replicate data among them, as they are accessed by the user. Changes are propagated opportunistically among the user's devices, and eventually pushed to the user's stationary computer (the *repository*). The repository keeps all versions of all files, so reconciliation of concurrently-updated copies of files is left to higher-level mechanisms, should it be needed. FEW [Preguica *et al.* 2005] takes a similar approach to automated file management for portable storage devices, but it provides automated reconciliation of diverging copies. FEW tracks the "causal history of update events that have been incorporated in the replica state" in order to ease reconciliation, e.g., using a three-way merge algorithm [Mens 2002]. The *Blue file system* takes a similar approach, with the aim of reducing the energy costs resulting from data accesses, and improving performance [Nightingale & Flinn 2004]. It achieves this by selecting nodes to read from and write to based on the expected energy costs and performance characteristics, in a way similar to Flashback [Loo *et al.* 2003] and OmniStore [Karypidis & Lalis 2005].

In *Bayou* [Demers *et al.* 1994], data replicas are stored on a set of *servers* (potentially stationary machines) while mobile nodes are assumed to access data by contacting neighboring servers, rather than locally replicating data. This architecture stemmed from the assumption that mobile nodes would not have sufficient storage capacities to store replicas, an assumption that no longer holds. All servers contributing to the store have an equivalent role: Bayou uses a *read-any/write-any* model, whereby mobile nodes can read from or write to any replica. Reconciliation and detection of update conflicts are handled by servers, in an application-specific way.

Several distributed file systems, such as Coda [Lee *et al.* 1999, Mummert *et al.* 1995], support *disconnected operation*, i.e., they allow clients to modify local copies of files when the server is unreachable. They follow a traditional client/server model: local modifications must eventually propagate to the server and, in the presence of concurrent updates, the server must reconcile them [Lee *et al.* 1999]; automatic reconciliation may fail, in which case manual intervention is needed. More generally, the file server is viewed as the authoritative source of files. Consequently, updates in such file systems are not atomic: an update is only "committed" when/if it reaches the server and is kept unmodified.

Likewise, mobile nodes in *AdHocFS* are assumed to replicate data from a *home server*, but they may modify them locally while disconnected [Boulkenafed & Issarny 2003]. Again, mobile nodes are expected to periodically synchronize with the home server, which may require reconciling divergent replicas or handling update conflicts; changes are only committed when they reach the server. However, AdHocFS also supports collaborative applications within *ad hoc* groups (e.g., among devices in vicinity of each other), such as collaborative document editing, without requiring access to the home server. It can provide *strong consistency* guarantees for such collaborative applications, e.g., through a simple locking scheme.

Haddock-FS goes one step further by employing a peer-to-peer, server-less approach [Barreto & Ferreira 2004]. It distinguishes between *tentative updates* (i.e., local changes that have not been propagated and reconciled), from *stable updates* (i.e., updates that have been propagated, reconciled and "committed" by a *primary* replica). Secondary replicas should update to the latest stable values.

The concerns of writable mobile storage are different from those of a backup system. A lot of complexity is added as a consequence of supporting concurrent updates of the data, which requires suitable update propagation and conflict resolution. In the end, work on mobile storage appears to have a lot in common with distributed revision control systems such as GNU Arch and Git [Lord 2005, Hamano 2006]. Such tools allow people working in a loosely connected fashion to commit their own changes locally, while still being able to periodically integrate updates made by others in their local copy. Merging is facilitated by tracking the set of changes already incorporated in each copy [Hamano 2006], in a way that is similar to the one used by, e.g., FEW [Preguica *et al.* 2005].

It is also worth noting that none of the aforementioned systems addresses cooperation among mutually suspicious participants in detail.

2.4.4. Delay-Tolerant Networking

delay-tolerant networks

Handling intermittent infrastructure connectivity by relaying data replicas through intermediate mobile nodes all the way to an on-line server makes our approach comparable to *delay-tolerant networks* (DTNs): data blocks that are transmitted by data owners to contributors can be viewed as packets sent to the Internet-based store and contributors can be viewed as relays [Zhang 2006]. The whole backup process can be viewed as a high-latency communication channel used to transfer snapshots from the data owner to the on-line store.

Recently, a large amount of work has gone into the design and evaluation of protocols for DTNs in general, and intermittently-connected mobile networks (ICMNs) in particular (sometimes called delay-tolerant mobile networks, or DTMNs). DTNs and ICMNs can be seen as (mobile) networks with a high probability of network partitioning that makes end-to-end connections highly unlikely. Such networks are expected to be common in remote regions, as well as in situations such as disaster relief efforts, interplanetary communication, sensor networks, and wildlife tracking [Fall 2003, Harras *et al.* 2007, Juang *et al.* 2002]—most of which are also relevant to our cooperative backup service. Like MANETs, DTNs rely on cooperative message routing.

To overcome the intermittent connectivity patterns found in those scenarios, DTNs aim to provide a protocol layer specifically tailored to cope with these characteristics. DTNs were first described as an *asynchronous* message delivery system, akin to email, where messages are routed across *network regions* through *DTN gateways* [Fall 2003]. Each networking region is characterized by a specific set of protocols (e.g., Internet-like, mobile *ad hoc*, etc.) and gateways act as bridges among them. Gateways typically provide *store-and-forward* functionality: they store messages coming from one region and "transport" them to the destination region. Transport could involve *physical* transportation, for instance by carrying a message to the target network.

The *ParaNets* approach extends this model by allowing different networking technologies to be used in parallel to improve the DTN performance [Harras *et al.* 2007]. The ParaNets design was motivated by the observation that diverse wireless communication technologies are now available, as was discussed in Section 2.1. Its authors propose to leverage this diversity by choosing an appropriate transport layer depending on the message type (e.g., using satellite communications for acknowledgments and short messages and using regular DTN message routing for larger messages).

Part of the work on DTNs focused on defining protocols by which mobile nodes can exchange data [Scott & Burleigh 2007, Cerf *et al.* 2007], and a large part of the work went into devising and evaluating routing algorithms [Zhang 2006, Spyropoulos *et al.* 2007]. Routing algorithm designers are primarily concerned with reducing transmission delays while keeping the resource requirements low.

Wang *et al.* summarize existing packet forwarding algorithms as shown on Figure 4. At one end of the spectrum, a naïve strategy consists in having the source node carry its data

Algorithm	Who	When	To Whom
flooding	all nodes	new contact	all new
direct contact	source only	destination	destination only
simple replication, r	source only	new contact	*r* first contacts
history, r	all nodes	new contact	*r* highest ranked

Figure 4. Summary of various forwarding algorithms for DTNs [Wang *et al.* 2005a].

until it is in *direct contact* with the destination node. This approach requires few resources but may lead to high end-to-end delays. At the other end is *epidemic routing* or *flooding* where each node forwards all message (including messages on behalf of other nodes) to any other node encountered. This may yield smaller delays at the cost of significant bandwidth and storage overhead [Spyropoulos *et al.* 2007]. In between these two extremes, a variant of flooding is the popular probabilistic forwarding where nodes forward messages with a probability less than one (not represented here). Another approach is *simple replication* where the message is handed over by the source node to a fixed number *r* of relays, each of which can then deliver it only to the destination node. Lastly, in *history-based* approaches (sometimes called utility-based, or estimation-based), nodes make "informed" forwarding decisions based on past encounters [Spyropoulos *et al.* 2007, Liao *et al.* 2006, Juang *et al.* 2002] and forward packets only to the *r* highest ranked nodes. A wealth of more sophisticated forwarding strategies is also described in the literature [Wang *et al.* 2005a, Widmer & Boudec 2005].

Interestingly, the "simple replication" approach to message routing is similar to the basic replication strategy we envision for our cooperative backup service: each data owner distributes copies of each data item to a limited number of contributors, each of which may in turn forward it directly to the Internet store (see Section 2.3.1). Similarly, the utility-based routing approach is comparable to the contributor selection process used by the Flashback [Loo *et al.* 2003] and OmniStore [Karypidis & Lalis 2005] cooperative stores for PAN.

However, delay-tolerant networking also differs from our cooperative backup approach in various ways. First, routing algorithms for DTNs usually assume that the destination of a message is a specific physical node. Conversely, the destination of data items in our approach is an Internet-accessible store reachable from a very wide range of physical locations (Internet access points), making it possible to hope for reasonable delays in only two "DTN hops". More importantly, *versioning* of the backed-up data makes a significant difference. In our cooperative backup service, several versions of the user's data may eventually reach the on-line store. Should the latest version be unavailable, being able to retrieve prior versions *is* valuable, from a fault-tolerance viewpoint, whereas a pure networking approach with no knowledge of messages semantics treats all messages equally.

From a security perspective, little work has been done to address issues arising from cooperation with untrusted principals in DTNs [Fall 2003, Farrell & Cahill 2006]. Our work contributes to this area, as will be discussed in Chapter 5.

2.5. Summary

The contributions of this chapter can be summarized as follows:

- We illustrated the need for improved fault-tolerance mechanisms for mobile devices, specifically focusing on the availability of data produced and carried by mobile devices.
- We defined *dependability goals* in terms of data availability, integrity and confidentiality.

In addition, we outlined a *cooperative backup* approach to tackle this issue:

- The proposed approach leverages *spontaneous interactions* among mobile devices by means of short-range wireless communications.
- Participating devices share storage resources and thereby cooperatively implement a distributed store that is used for backup purposes.
- Mobile devices storing data on behalf of other nodes eventually send them to an agreed-upon store on the Internet.
- Data recovery works by querying this on-line store for data backups.

Finally, we studied related work in the areas of checkpointing and rollback, Internet-based peer-to-peer cooperative storage, distributed mobile storage, and delay-tolerant networking.

Chapter 3. Analytical Evaluation of the Proposed System

This chapter provides an analytical evaluation of the dependability gains to be expected from our cooperative backup service, as described in Section 2.3. Some of the results presented here were described in Ossama Hamouda's Master Thesis [Hamouda 2006] and in [Courtès *et al.* 2007a].

3.1. Introduction

Various replication and data scattering algorithms may be used to implement the cooperative backup service. Replication may be handled by creating full copies of individual data items (we refer to this as *simple replication*) or by more sophisticated *erasure coding* techniques. Choosing erasure codes allows an increase of data fragmentation and, subsequently, data dissemination. Increasing fragmentation and dissemination is beneficial from a confidentiality viewpoint [Deswarte *et al.* 1991]; however, its impact on data dependability, particularly in our scenario, is unclear. The analytical evaluation presented in this chapter aims to clarify this.

Furthermore, the effectiveness of the proposed cooperative backup service, from a fault-tolerance viewpoint, depends on a number of environmental factors: it should provide most benefit with a relatively high density of participating devices and intermittent Internet access for all devices, but also needs to account for occasional device failures and potentially malicious contributor behavior. In order to gain confidence about the efficiency of our backup service, we need to assess the impact of these issues on data dependability.

We analyze the fault-tolerance gain provided by MoSAIC as a function of (i) the various environmental parameters (frequency of Internet access, contributor encounter rate, node failure rate) and (ii) different replication strategies. Our approach is based on model-based evaluation, which is well suited to support design tradeoff studies and to analyze the impact of several parameters of the design and the environment from the dependability and performance points of view. We expect such an analysis to provide us with a better understanding of the dependability gains to be expected from our service.

We identify two main goals. First, the analysis should help us determine under what circumstances MoSAIC is the most beneficial, compared to solutions that do not replicate data in the *ad hoc* domain. Second, it should help us choose among different replication strategies, depending on a given scenario's parameters and user preferences (e.g., target data availability, confidentiality requirements).

The major contribution of the work presented here lies in the *assessment* of data replication and scattering strategies using analytical methods, taking into account a variety of influential parameters. It differs substantially from earlier evaluation work by other authors due to the entirely novel characteristics of the system and environment modelled (see Section 3.5).

Section 3.2 provides background information on erasure codes. Section 3.3 describes our methodology. Section 3.4 summarizes the results obtained and discusses their impact on the design of the cooperative backup service. Section 3.5 presents related work. Finally, Section 3.6 summarizes our findings.

3.2. Background

Erasure coding algorithms have been studied extensively [Lin *et al.* 2004, Mitzenmacher 2004, Xu *et al.* 1999, Xu 2005, Weatherspoon & Kubiatowicz 2002]. Here we do not focus on the algorithms themselves but on their properties. A commonly accepted definition of erasure coding is the following [Xu 2005, Lin *et al.* 2004]:

- Given a k-symbol input datum, an erasure coding algorithm produces $n \geq k$ *fragments.*
- Any m fragments are necessary and sufficient to recover the original datum, where $k \leq m \leq n$. When $m = k$, the erasure code algorithm is said to be *optimal* [Xu *et al.* 1999].

Figure 5 illustrates the coding and recovery processes with an optimal erasure code. Although not all erasure coding algorithms are optimal (many of them are *near-optimal* [Xu *et al.* 1999]), we will assume in the sequel the use of an optimal erasure code where $m = k$. By convention, we note (n,k) such an optimal erasure code [Xu 2005].

When all k fragments are stored on different devices, an optimal erasure code allows $n - k$ failures (or erasures) to be tolerated (beside that of the primary replica). The storage cost (or *stretch factor*) for an optimal erasure code is $\frac{n}{k}$ (the inverse ratio $\frac{k}{n}$ is often called the *rate* of an erasure code). To tolerate a number of erasures f, we need $n = k + f$, so the storage cost is $1 + {}^{f}\!/_{k}$. Therefore, erasure coding (with $k \geq 2$) is more storage-efficient than simple replication ($k = 1$). For instance, (2,1) and (3,2) erasure codes can both allow the tolerance of one failure, but the former requires twice as much storage as the original data while the latter only requires 1.5 times as much.

Additionally, when all k fragments are distributed to different devices belonging to different non-colluding users (or under different administrative domains), erasure codes can be regarded as a means for improving data confidentiality: to access the data, an

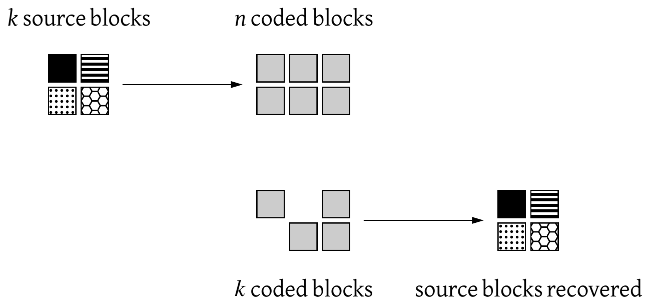


Figure 5. Coding and recovery with an optimal erasure code, with $k = 4$ and $n = 6$.

attacker must have control over k contributing devices instead of just one when simple replication is used [Deswarte *et al.* 1991]. This effectively raises the bar for confidentiality attacks and may usefully complement ciphering techniques used at other layers. Similar concerns are addressed by generalized threshold schemes where, in addition to the definition above, less than $p \leq k$ fragments convey no information about the original data, from an information-theoretic viewpoint [Ganger *et al.* 2001].

3.3. Methodology

In this section, we present the approach we have taken to model and evaluate our cooperative backup service. In particular, we describe and discuss the characteristics of the system modeled. We then present our use of Markov chains and the evaluated dependability measures.

3.3.1. System Characteristics

The cooperative backup service that we model is characterized by its replication and scattering strategy, and the considered device-to-device and device-to-Internet backup opportunities.

3.3.1.1. Replication Strategy

Our model considers the case of a data owner that needs to replicate a single data item (generalization of the results to more than one data item is straightforward). We consider that the owner follows a *pre-defined* replication and dissemination strategy, using (n,k) erasure coding, where n is decided off-line, *a priori*, and where the owner distributes one and only one fragment to each encountered contributor. When $k = 1$, the strategy corresponds to simple replication. In practice, the exact choice of n and k could be made as a function of the currently-perceived backup opportunity rates, and the user's dependability and confidentiality requirements.

This replication strategy privileges confidentiality over backup reliability: only one fragment is given to each contributor encountered[1], at the risk of being unable to distribute all the fragments in the end (for instance, because not enough contributors are encountered). An alternative strategy that favors backup reliability over confidentiality consists in providing a contributor with as many fragments as possible while it is reachable.

Furthermore, the replication strategy is considered *static*. In particular, we consider that owners are not aware of the failures of contributors storing data on their behalf. Thus, owners cannot, for instance, decide to create more replicas when previously encountered contributors have failed. The fact that owners cannot be made aware of contributor failures is realistic under most of the scenarios envisaged: contributors are likely to be out of reach of owners at the time of failure and, in addition, it is impossible to distinguish between a slow and a failed node [Fischer *et al.* 1985].

3.3.1.2. Backup Opportunities

We consider that every encounter between devices offers a device-to-device backup opportunity. Specifically, every device encountered is considered to be a contributor that *unconditionally accepts* storage requests from the data owner. Data owners unconditionally send one data fragment to each contributor encountered. Later, in Section 3.4.5, we also consider scenarios where more than one data fragment is sent during a contributor encounter. Note that scenarios in which not all encounters offer backup opportunities (e.g., with contributors refusing to cooperate) can be simply modeled by introducing an opportunity/encounter ratio as an additional parameter. In other words, trust and cooperation among devices are not modeled but these topics are addressed in Chapter 5.

We consider that Internet connection is only exploited when it is cheap and provides a high bandwidth. Furthermore, we consider that every such device-to-Internet backup opportunity is exploited to the full, i.e., whenever a node gains Internet access, we assume that it transfers all the data fragments it currently stores on behalf of other nodes.

3.3.2. Modeling Approach

Two complementary techniques can be used to support model-based evaluation approaches: analytical techniques and simulation. Analytical techniques are commonly used to support dependability evaluation studies. They allow one to obtain mathematical expressions of the relevant measures, which can then be explored to easily identify trends and to carry out sensitivity analysis. When using analytical techniques, the system must be described at a high level of abstraction. Simplifying assumptions are generally needed to obtain tractable models. Although simulation can be used to describe the system at a more

[1] According to this policy, a contributor encountered more than once will only be given a fragment the first time it is encountered.

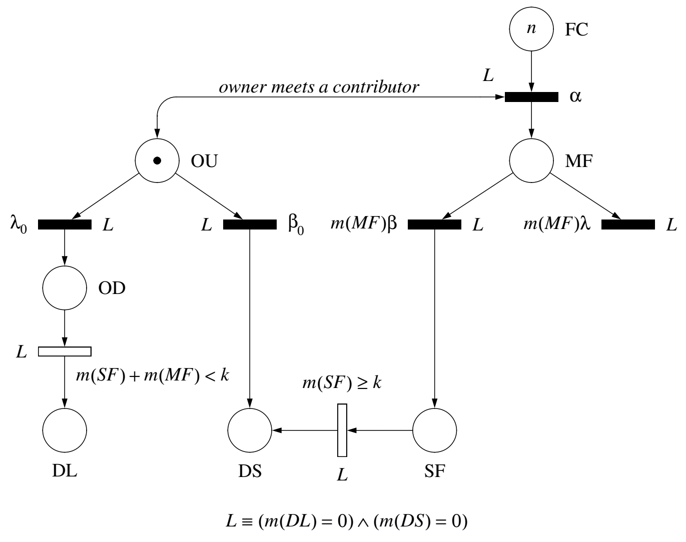


Figure 6. Petri net of the replication and scattering process for an (*n*,*k*) erasure code.

detailed level, it is more costly in terms of the processing time needed to obtain accurate and statistically significant quantitative results.

Markov chains and generalized stochastic Petri nets (GSPNs) are commonly used to perform dependability evaluation with sensitivity analyses aimed at identifying parameters having the most significant impact on the measures. The corresponding models are based on the assumption that all the underlying stochastic processes are described by exponential distributions. Although such an assumption may not faithfully reflect reality, the results obtained from the models and sensitivity analysis give preliminary indications and estimations about the expected behaviors and trends that can be observed. More accurate results can be obtained considering more general distributions, using for example the "stages method" [Cox & Miller 1965] or non Markovian models. However, in this chapter, we assume that all stochastic processes are exponentially distributed.

In the following, we present a generic GSPN model and the corresponding Markov chains. Then, we present the quantitative measures evaluated from the models to assess data dependability. Finally, we discuss the main parameters that are considered in the sensitivity analysis studies.

3.3.3. GSPN and Markov Models

GSPNs are widely used to support the construction of performance and dependability evaluation models based on Markov chains. In particular, they are well suited to represent synchronization, concurrency and dependencies among processes [Marsan *et al.* 1995].

Figure 6 presents the GSPN model of MoSAIC using an (*n*,*k*) erasure coding algorithm.

The model focuses on the mobile *ad hoc* part of the cooperative backup service, purposefully ignoring issues related to the implementation of the Internet-side functionalities. Thus, a data fragment is considered "safe" (i.e., it cannot be lost) whenever either its owner or a contributor storing it is able to access the Internet. In other words, the Internet-based store of our cooperative backup service is abstracted as a "reliable store". Conversely, if a participating device fails before reaching the Internet, then all the fragments it holds are considered lost.

Thus, with (*n*,*k*) erasure coding, a data item is definitely lost if and only if its owner device fails *and* less than *k* contributors hold or have held a fragment of the data item.

Our model consists of three main processes represented by timed transitions with constant rate exponential distributions:

- A process with rate α that models the encounter of a contributor by the data owner, where the owner sends one data fragment to the contributor.
- A process that models the connection of a device to the Internet, with rate β_0 for the owner and β for contributors.
- A process that represents the failure of a device (crash fault), with rate λ_0 for the owner and λ for contributors.

The GSPN in Figure 6 is divided into two interacting subnets. The subnet on the left describes the evolution of a data item at the owner device: either it is lost (with rate λ_0), or it reaches the on-line reliable store (with rate β_0). Places OU and OD denote situations where the owner device is "up" or "down", respectively. The subnet on the right describes: (i) the data replication process leading to the creation of "mobile fragments" (place MF) on contributor devices as they are encountered (with rate α), and (ii) the processes leading to the storage of the fragments (place SF) in the reliable store (rate β), or their loss caused by the failure of the contributor device (rate λ). At the top of the right-hand side subnet is place FC whose initial marking denotes the number of fragments to create. The transition rates associated with the loss of a data fragment or its storage on the Internet are weighted by the marking of place MF (denoted $m(MF)$), i.e., the number of fragments that can enable the corresponding transitions.

Two places with associated immediate transitions are used in the GSPN to identify when the data item is safely stored in the reliable store (place DS), or is definitely lost (place DL), respectively. The "data safe" state is reached (i.e., DS is marked) when the original data item from the owner node or at least *k* fragments from the contributors reach the Internet. The "data loss" state is reached (i.e., DL is marked) when the data item from the owner node is lost and less than *k* fragments are available. This condition is represented by a predicate associated with the immediate transition that leads to DL. Finally, L is the GSPN "liveliness predicate", true if and only if $m(DS) = m(DL) = 0$: as soon as either DS or DL contains a token, no transition can be fired.

The GSPN model of Figure 6 is generic and can be used to automatically generate the Markov chain associated with any (n,k) erasure code. As an example, Figure 7 shows the Markov chain representing replication and scattering with a (2,1) erasure code (i.e., simple replication). The edge labels represent the rate of the corresponding transition; for instance, an edge labeled "2λ" represents the failure of one of the two contributors holding a copy of the owner's data. On this graph, we distinguish four sets of states:

1. The states labeled "alive(X) Y/Z" denote those where the owner device is up, where X, Y and Z represent, respectively, the number of fragments left to distribute, the number of fragments held by contributors, and the number of fragments available in the on-line reliable store (i.e., "safe"). Here, $k \leq X + Y + Z \leq n$ and $Z < k$ (for the (2,1) erasure code presented on Figure 7, Z is always zero since $Z = k = 1$ is equivalent to the "data safe" state).
2. States labeled "dead Y/Z" represent those where the owner device has failed but where backup can still succeed thanks to contributors. Here, $Y + Z \geq k$ and $Z < k$.
3. The state labeled "alive/endangered" aggregates all states where the owner device is available and where $X + Y + Z < k$. In this situation, although fragments may be available in the reliable store and on contributing devices, failure of the owner device results in definite data loss.
4. The two absorbing states, labeled "DS" ("data safe") and "DL" ("data lost"), represent, respectively, the safe arrival of the data item in the reliable store and its definite loss.

Similar loop-free Markov chains can be generated for any (n,k). The total number of states in such an (n,k) Markov chain is $O(n^2)$. The models we are considering, with reasonably small values of n (i.e., small storage overhead), are tractable using available modeling tools.

It should be clear from the above examples that a *data forwarding* strategy implemented by contributors (see Section 2.3.1) would not bring any dependability benefit. Data block forwarding would be represented as a transition from each state where the number of fragments held by contributors is non-zero to itself, which would not change the asymptotic probability of reaching DS. A strategy where contributors would not only forward but also replicate data would certainly improve data dependability. However, for reasons exposed in Section 2.3.1, we decided not to explore such flooding strategies.

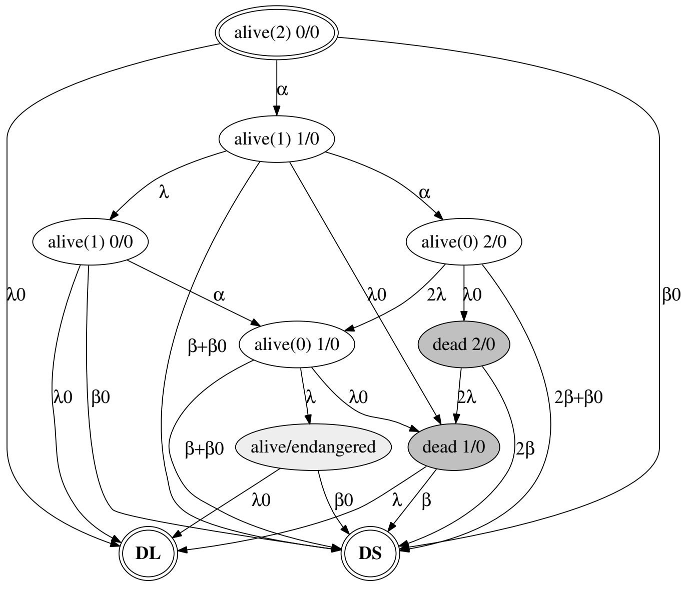


Figure 7. Markov chain of the replication and scattering process for a (2,1) erasure code.

3.3.4. Generating Markov Chains from Production Rules

During the early stages of this evaluation effort, we implemented a tool able to generate Markov chains representing the replication strategies of interest. This tool is implemented in Scheme [Kelsey *et al.* 1998], using GNU Guile [Jaffer *et al.* 1996]. The Markov chain states as manipulated by the program are labeled in a way similar to the one presented above. A set of simple state/transition *production rules* makes use of this information to create state objects and transitions among them. Being closely related to the process being modeled, this programming framework allowed us to quickly gain a better understanding of our model. This approach also permitted incremental experimentation, starting from an implementation of the simple replication strategy (i.e., $k = 1$) and augmenting it to handle the more general erasure coding case.

Figure 8 shows the Scheme code modeling simple replication[2]. For a given Markov chain state, this code produces all the relevant transitions to other states by mutating the current list of transitions associated with the state[3]. All states are labeled by a 3-element

[2] Note that Scheme derives from Lisp and, as such, it uses a prefix notation. For example, the application of function f to arguments x and y, which mathematicians note $f(x,y)$, is noted `(f x y)`. Basic mathematical operators are treated like other functions, so $x + y$ is noted `(+ x y)`. By convention, functions whose names end with an exclamation mark denote functions with side-effects, i.e., functions that modify their argument. Also note that `#t` and `#f` denote “true” and “false”, respectively.

[3] This approach follows an imperative programming paradigm, relying on side-effects to modify state objects. Since our Markov chain is loop-free, it would be possible to create all the states and transitions in a purely

```
;; contributor failure
(if (<= avail 1)
    (push-to-ko! 'lambda)
    (push! (make-label alive? (- avail 1) to-do)
           `(* ,(if alive? (- avail 1) avail) lambda)))

;; internet access
(if (>= avail 1)
    (push-to-ok! `(* beta ,avail)))

(if alive?
    (begin
      ;; contributor encounter
      (if (>= to-do 1)
          (push! (make-label #t (+ avail 1) (- to-do 1)) 'alpha))

      ;; owner failure
      (if (= avail 1)
          (push-to-ko! 'lambda0)
          (push! (make-label #f (- avail 1) to-do) 'lambda0))))
```

Figure 8. Transition production rules for the simple replication strategy.

vector containing (i) a boolean denoting whether the owner device is up, and (ii) the Y and Z integers as presented in Section 3.3.3. Such labels are constructed by invoking the `make-label` procedure and passing it these 3 elements. Procedure `push!` creates a transition from the current state to the state labeled as specified as the first parameter, and with the rate specified as the second parameter; similarly, `push-to-ok!` creates a transition to the DS state, while `push-to-ko!` creates a transition to the DL state. Variables `to-do` and `avail` represent the number of replicas that must be made and the number of replicas already available, respectively; `alive?` is a boolean indicating whether the current state denotes a situation where the owner device is available. The generalized production rules for erasure codes are similar but address more cases.

This tool allowed us to generate graphical representations of our Markov chains, using Graphviz package for automated graph layout [Gansner *et al.* 1993], as shown in Figure 7 and Figure 18. This proved a useful model debugging aid. Our tool is also capable of computing asymptotic probability of a successful backup on our absorbing Markov chains (both symbolically and numerically[4]); it can produce plotting data that can be fed to GNUplot to produce plots such as those presented in this chapter.

We complemented this approach with the SURF-2 dependability evaluation tool which can additionally handle Petri nets [Béounes *et al.* 1993].

functional way, i.e., without modifying objects. Nevertheless, this imperative approach allows more general cases to be addressed.

[4] For symbolic computations, the raw, non-simplified expressions are returned. Simplification is left to an external computer algebra system such as Axiom or Maxima.

3.3.5. Quantitative Measures

We analyze the dependability of our backup service *via* the probability of data loss, i.e., the asymptotic probability, noted PL, of reaching the "data lost" state. For a given erasure code (n,k), PL can be easily evaluated from the corresponding Markov chain using well-known techniques for absorbing Markov chains [Kemeny & Snell 1960]. The results shown in the next section were obtained using symbolic computations of PL produced by the tool presented above. The smaller PL is, the more dependable is the data backup service.

To measure the dependability improvement offered by MoSAIC, we compare PL with the probability of data loss PL_{ref} of a comparable, non-MoSAIC scenario where:

- data owners do not cooperate with other mobile devices;
- data owner devices fail with rate λ_0;
- data owners gain Internet access and send their data items to a reliable store with rate β_0.

This scenario is modeled by a simple Markov chain where the owner's device can either fail and lose the data, or reach the Internet and save the data. The probability of loss in this scenario is: $PL_{ref} = \frac{\lambda_0}{\lambda_0+\beta_0}$.

We note LRF the data *loss probability reduction factor* offered by MoSAIC compared to the above non-MoSAIC scenario, where $LRF = PL_{ref}/PL$. The higher LRF, the more MoSAIC improves data dependability. For instance, $LRF = 100$ means that data on a mobile device is 100 times more unlikely to be lost when using MoSAIC than when not using it.

3.3.6. Parameters

PL and LRF depend on a number of parameters (n, k, α, β, λ, β_0, and λ_0). Rather than considering absolute values for the rates of stochastic processes, we consider ratios of rates corresponding to pertinent competing processes.

For example, the usefulness of cooperative backup will depend on the rates at which contributing devices are met relative to the rate at which connection to the fixed infrastructure is possible. Therefore, it makes sense to study LRF as a function of the *ad hoc*-to-Internet *connectivity ratios* $\frac{\alpha}{\beta}$ and $\frac{\alpha}{\beta_0}$ rather than for absolute values of these parameters.

connectivity ratios

Similarly, the effectiveness of the contributors and the data owner towards data backup depends on the rate at which they are able to connect to the Internet relative to the rate at which they fail. We thus define the ratios $\frac{\beta}{\lambda}$ and $\frac{\beta_0}{\lambda_0}$ as the *effectiveness* of the contributors and the data owner. We will study the evolution of dependability improvement when the effectiveness of contributors varies relative to that of the data owner. This allows us to study the impact of contributor ineffectiveness (leading to $\lambda > \lambda_0$). For most measures, however, we will assume that $\beta = \beta_0$ and $\lambda = \lambda_0$.

Finally, one may question the assumption that contributors accept *all* requests, at rate α, regardless of their amount of available resources. However, simple back-of-the-envelope calculations provide evidence that this is a reasonable assumption. When the replication strategy described in Section 3.3.1.1 is used, the number of fragments (i.e., storage requests) that a contributor may receive during the time between two consecutive Internet connections is, on average, $\frac{\alpha}{\beta}$. Let s be the size of a fragment: a contributor needs, on average, $V = s\left(\frac{\alpha}{\beta}\right)$ storage units to serve all these requests. If a contributor's storage capacity, C, is greater than V, it can effectively accept all requests; otherwise, the contributor is *saturated* and can no longer accept any storage request.

In other words, redefining α as the *effective* encounter rate (i.e., the rate of encounters of contributors that accept storage requests), and letting γ be the *actual* encounter rate, we have: $\frac{\alpha}{\beta} = \min\left(\frac{\gamma}{\beta}, \frac{C}{s}\right)$. A realistic estimate with $C = 2^{30}$ (contributor storage capacity of 1 GB) and $s = 2^{10}$ (fragment size of 1 KB) shows that contributors would only start rejecting requests when $\frac{\gamma}{\beta} > 2^{20}$, a ratio that is beyond most realistic scenarios.

3.4. Results

This section presents and discusses the results of our analysis.

3.4.1. Overview

We first assume that contributors and owners behave identically, i.e., $\beta_0 = \beta$ and $\lambda_0 = \lambda$.

Figure 9 shows the absolute probability of data loss without MoSAIC (PL_{ref}, as defined earlier), and with MoSAIC, using various erasure code parameters. In the MoSAIC case, the *ad hoc*-to-Internet connectivity ratio $\frac{\alpha}{\beta}$ is fixed at 100. Both with and without MoSAIC, the probability of loss falls faster. For example, with MoSAIC, the probability of data loss falls to zero as $\frac{\beta}{\lambda}$ tends to infinity; however, using MoSAIC, the probability of loss falls below 0.01 with $\frac{\beta}{\lambda}{\approx}6$, while it requires $\frac{\beta}{\lambda}{\approx}100$ to reach the same value without MoSAIC.

Figure 10 shows the loss reduction factor (i.e., the data dependability improvement) yielded by MoSAIC with a (2,1) erasure code using the replication strategy outlined in Section 3.3.1.1. Three observations can be made from this plot.

First, as expected, the cooperative backup approach is not very relevant compared to the reference backup approach when $\frac{\alpha}{\beta} = 1$ (i.e., when Internet access is as frequent as *ad hoc* encounters). Figure 11 shows the contour lines of LRF extracted from Figure 10: for the

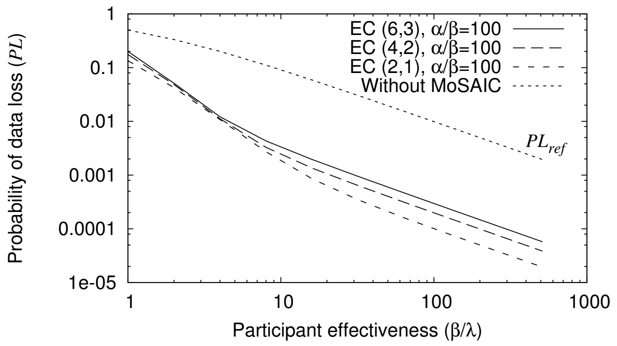


Figure 9. Data loss probability with and without MoSAIC.

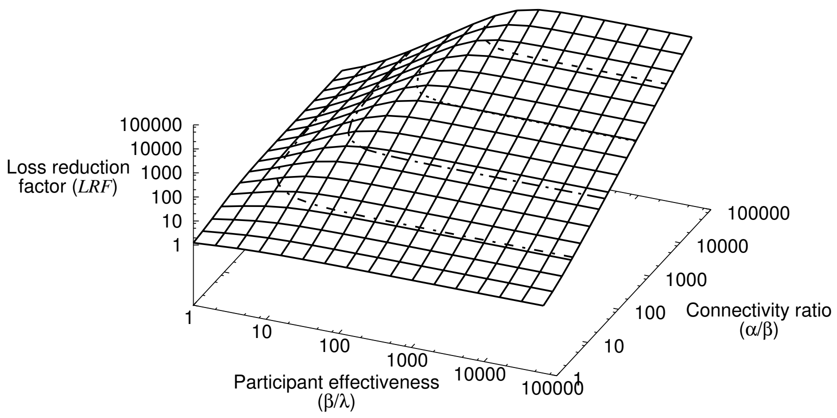


Figure 10. Loss reduction factor *LRF* for a (2,1) erasure code.

cooperative backup approach to offer at least an order of magnitude improvement over the reference backup scheme, the environment must satisfy $\frac{\beta}{\lambda} > 2$ and $\frac{\alpha}{\beta} > 10$.

Second, for any given $\frac{\alpha}{\beta}$, *LRF* reaches an asymptote after a certain $\frac{\beta}{\lambda}$ threshold. Thus, for any given connectivity ratio $\frac{\alpha}{\beta}$, increasing the infrastructure connectivity to failure rate ratio $\frac{\beta}{\lambda}$ is only beneficial up to that threshold.

Third, the dependability improvement factor first increases proportionally to $\frac{\alpha}{\beta}$, and then, at a certain threshold, rounds off towards an asymptote (visible on Figure 10 for small values of $\frac{\beta}{\lambda}$ but hidden for high values due to choice of scale). Other (n,k) plots have a similar shape.

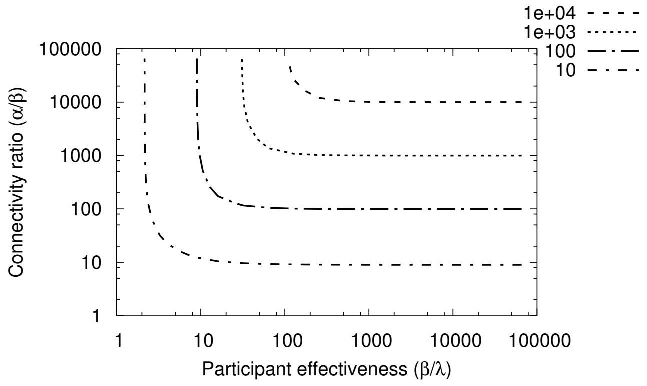


Figure 11. Contour map of *LRF* for Figure 10.

3.4.2. Asymptotic Behavior

Figure 12 shows *LRF* as a function of $\frac{\beta}{\lambda}$, for different values of $\frac{\alpha}{\beta}$ and different erasure codes (again, assuming the data owner's failure and connection rates are the same as those of contributors). This again shows that the maximum value of *LRF* for any erasure code, as $\frac{\beta}{\lambda}$ tends to infinity, is a function of $\frac{\alpha}{\beta}$. Using the symbolic output generated by our evaluation tool (see Section 3.3.4), we verified the following formula for a series of codes with $n \in \{2,3,4,5\}$ and $k \in \{1,2,3\}$ (with $k \leq n$) and postulate that it is true for all positive values of n and k such that $n \geq k$:

$$\lim_{\frac{\beta}{\lambda}\to\infty}\left(LRF_{n,k}\left(\frac{\alpha}{\beta},\frac{\beta}{\lambda}\right)\right) = \frac{\left(1+\frac{\alpha}{\beta}\right)^k}{\sum_{x=0}^{k-1}\binom{k}{x}\left(\frac{\alpha}{\beta}\right)^x} \tag{3.1}$$

This equation can be simplified as follows:

$$\lim_{\frac{\beta}{\lambda}\to\infty}\left(LRF_{n,k}\left(\frac{\alpha}{\beta},\frac{\beta}{\lambda}\right)\right) = \frac{\left(1+\frac{\alpha}{\beta}\right)^k}{\left(1+\frac{\alpha}{\beta}\right)^k-\left(\frac{\alpha}{\beta}\right)^k} = \frac{1}{1-\left(\frac{\frac{\alpha}{\beta}}{1+\frac{\alpha}{\beta}}\right)^k} \tag{3.2}$$

First, it describes an asymptotic behavior, which confirms our initial numerical observation. Second, it does not depend on *n*. This observation provides useful insight on how to choose the most appropriate erasure coding parameters, as we will see in Section 3.4.3.

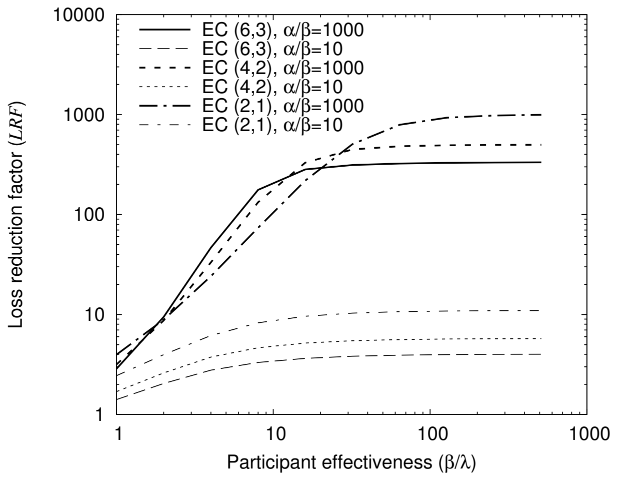


Figure 12. Loss reduction factor for different erasure codes.

We also computed the limiting value of $LRF(n, k)$ as $\frac{\alpha}{\beta}$ tends to infinity:

$$\lim_{\frac{\alpha}{\beta}\to\infty}\left(LRF_{n,k}\left(\frac{\alpha}{\beta},\frac{\beta}{\lambda}\right)\right) = \frac{\left(1+\frac{\beta}{\lambda}\right)^n}{\sum_{x=0}^{k-1}\binom{n}{x}\left(\frac{\beta}{\lambda}\right)^x} \tag{3.3}$$

This expression shows that LRF also reaches an asymptote as $\frac{\alpha}{\beta}$ grows, and that the value of this asymptote is dependent on $\frac{\beta}{\lambda}$.

3.4.3. Erasure Coding vs. Simple Replication

Figure 12 allows us to compare the improvement factor yielded by MoSAIC as different erasure codes are used. The erasure codes shown on the plot all incur the same storage cost: $\frac{n}{k} = 2$. In all cases, the maximum dependability improvement decreases as k increases. This is confirmed analytically by computing the following ratio, for any $p > 1$ such that pk and pn are integers:

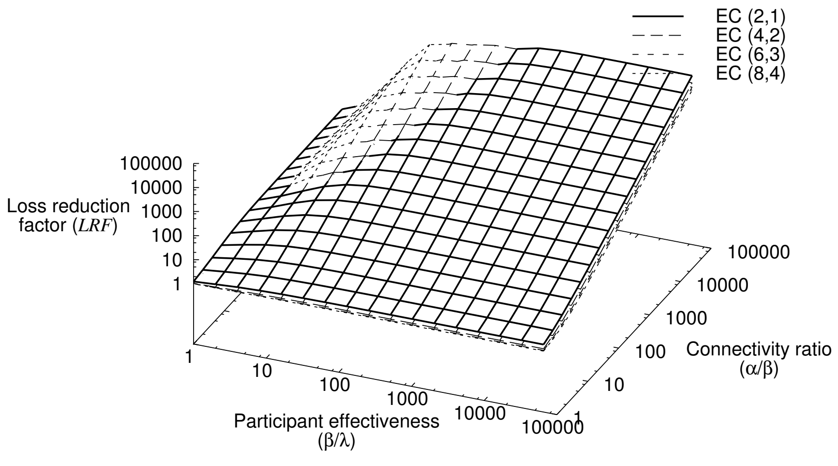


Figure 13. Comparing *LRF* for different erasure codes with $\frac{n}{k} = 2$.

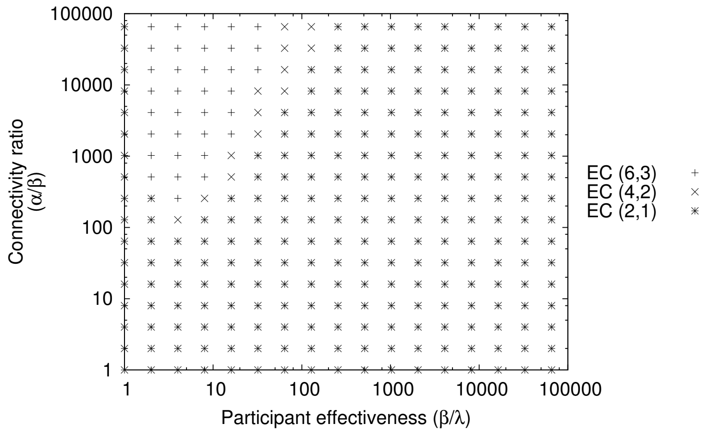


Figure 14. Comparing *LRF* for different erasure codes with $\frac{n}{k} = 2$: projection.

$$R_p = \frac{\lim\limits_{\frac{\beta}{\lambda}\to\infty}\left(LRF_{pn,pk}\left(\frac{\alpha}{\beta},\frac{\beta}{\lambda}\right)\right)}{\lim\limits_{\frac{\beta}{\lambda}\to\infty}\left(LRF_{n,k}\left(\frac{\alpha}{\beta},\frac{\beta}{\lambda}\right)\right)} = \frac{1-\left(\frac{\frac{\alpha}{\beta}}{1+\frac{\alpha}{\beta}}\right)^{k}}{1-\left(\frac{\frac{\alpha}{\beta}}{1+\frac{\alpha}{\beta}}\right)^{kp}} \tag{3.4}$$

We see that $R_p < 1$ for $p > 1$. Thus, we conclude that, from the dependability viewpoint, simple replication (i.e., with $k = 1$) is *always* preferable to erasure coding (i.e., with $k > 1$) above a certain $\frac{\beta}{\lambda}$ threshold.

Different trends can be observed for lower values of $\frac{\beta}{\lambda}$. As illustrated on Figures 12 and 13, we can numerically compare the dependability yielded by various erasure codes. Figure 13 compares the dependability improvement yielded by several erasure codes having the same storage cost; only the top-most erasure code (i.e., the surface with the highest LRF) is visible from above. The (2,1) plot is above all other plots, except in a small region where the other erasure codes (thin dashed and dotted lines) yield a higher LRF.

Figure 14, which is a projection of this 3D plot on the $\frac{\beta}{\lambda}$ and $\frac{\alpha}{\beta}$ plane, shows the region where erasure codes perform better than simple replication. Each point of the plot shows the erasure coding strategy that yields the highest LRF for the given $\frac{\alpha}{\beta}$ and $\frac{\beta}{\lambda}$ ratios. For instance, plus signs denote situations where a (6,3) erasure code provides a higher LRF than the two other erasure codes. We observe that simple replication (a (2,1) code) yields better dependability than erasure coding in a large spectrum of scenarios. Erasure codes yield a higher data dependability than simple replication in the region defined (roughly) by $\frac{\alpha}{\beta} > 100$ and $1 < \frac{\beta}{\lambda} < 100$. However, in this region, the dependability yielded by erasure codes is typically less than an order of magnitude higher than that yielded by simple replication, even for the (extreme) case where $\frac{\alpha}{\beta} = 1000$ (see Figure 12).

Interestingly, similar plots obtained for larger values of $\frac{n}{k}$ (e.g., see Figure 15) show that the region where erasure codes prevail tends to shift towards lower $\frac{\beta}{\lambda}$ values as $\frac{n}{k}$ increases. In other words, the spectrum of scenarios where erasure codes provide better dependability than simple replication narrows as the chosen storage overhead (the $\frac{n}{k}$ ratio) increases.

Nevertheless, when confidentiality is an important criterion, using erasure coding instead of simple replication is relevant. Erasure coding can achieve better confidentiality than simple replication [Deswarte *et al.* 1991] at the cost of a slightly lower asymptotic dependability improvement factor. For instance, in the context of Figure 12, if the user wants to maximize confidentiality while requiring a minimum improvement factor of 100, a (6,3) erasure code should be chosen rather than simple replication.

3.4.4. Impact of Contributor Effectiveness

We now consider the case where contributors and owners behave differently from a reliability viewpoint. The fact that contributors may be less reliable than the data owner can be studied by distinguishing the owner's Internet access and failure rates (β_0 and λ_0) from the contributor's (β and λ). Since the cooperative backup service is open to anyone willing to participate, participants do not trust each other *a priori*. It may be the case that some contributors are malicious or otherwise largely unreliable, thereby proving to be

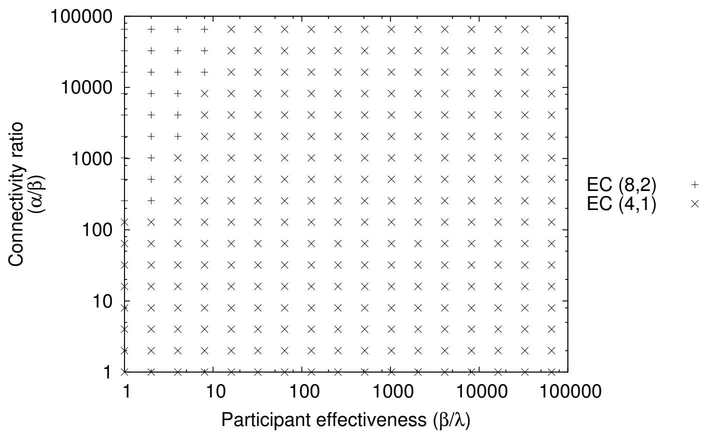


Figure 15. Comparing *LRF* for different erasure codes with $\frac{n}{k} = 4$: projection.

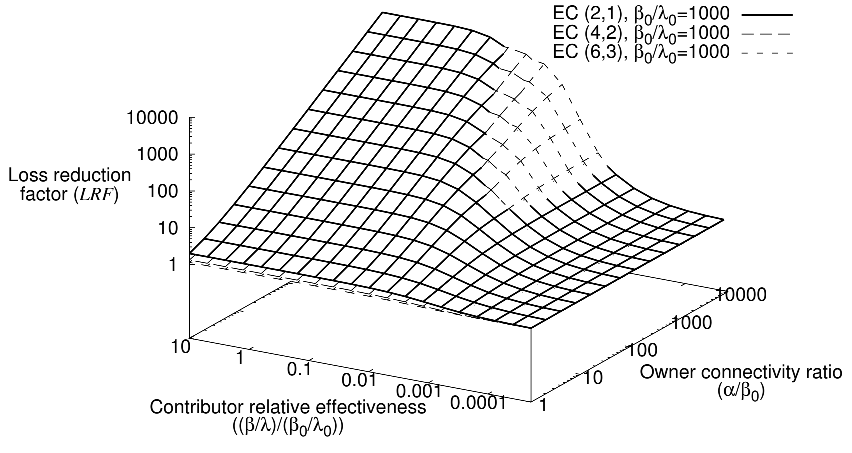


Figure 16. Loss reduction factor as contributors become less effective.

less *effective*, with respect to accessing the on-line reliable store, than the data owner, as mentioned in Section 3.3.6. We define the *effectiveness* of a contributor (respectively, the data owner) as a backup agent by the ratio $\frac{\beta}{\lambda}$ (respectively $\frac{\beta_0}{\lambda_0}$). Thus, the effectiveness of a contributor relative to that of the data owner can be expressed by $\frac{\beta/\lambda}{\beta_0/\lambda_0}$.

Figure 16 shows *LRF* as a function of the contributor-to-owner effectiveness ratio on one hand, and the owner's *ad hoc* to Internet connectivity ratio on the other. At the left end of the spectrum, where contributors are as effective or even more so than the owner itself, MoSAIC is very beneficial in terms of backup dependability. In this scenario, the data loss reduction improvement is proportional to $\frac{\alpha}{\beta_0}$ up to an asymptote, as was already discussed in the previous sections.

Not surprisingly, at the other end of the spectrum, the curve shows that as contributors become less effective than the data owner, LRF decreases until it becomes equivalent to the non-MoSAIC case (i.e., $LRF = 1$). More precisely, when contributors become 100 times less effective than the owner, the loss reduction improvement factor yielded by MoSAIC is less than 10. The threshold at which this occurs does not depend on the value of $\frac{\alpha}{\beta_0}$.

Furthermore, Figure 16 shows that the threshold is the same for all erasure codes. However, again, there is a small region of the plot where erasure codes are more resilient against contributor ineffectiveness than simple replication. For this setup, the region where erasure codes prevail over simple replication is limited to effectiveness ratios between 0.1 and 0.001 and $\frac{\alpha}{\beta_0}$ above 100. With $\frac{\alpha}{\beta_0} = 1000$, the gain provided by erasure codes in this region is less than an order of magnitude, as shown on Figure 17. Thus, for many use cases of MoSAIC, using erasure coding instead of simple replication will not provide appreciable improvement in the face of misbehaving contributors, from the dependability point of view.

3.4.5. Hybrid Scattering Strategies

We augmented our model to allow the evaluation of scattering strategies where more than one fragment is transferred during a peer encounter. We refer to them as "hybrid" scattering strategies since they are a middle-ground between the erasure-code and simple replication strategies evaluated earlier. Doing so requires small changes to the set of Markov chain production rules and/or to the Petri net presented in Figure 6. A new parameter, t, denotes the number of fragments transferred per contributor encounter. Scenarios presented earlier correspond to $t = 1$. When $n \bmod t$ is non-zero, the last contributor encountered is passed only $n \bmod t$ fragments instead of t; likewise, if this last-encountered contributor fails (respectively, accessed the Internet), then only $n \bmod t$ fragments are lost (respectively, saved). Figure 18 illustrates these situations with a (5,3) code and $t = 2$: from the "alive(0) 5/0" state, either the last contributor fails (leading to state "alive(0) 4/0") or one of the other two contributors fails (leading to state "alive(0) 3/0"), likewise for Internet access.

Figure 19 shows the impact of sending more than one fragment per connection on LRF. Not surprisingly, (6,3) with $t = 3$ and (2,1) with $t = 1$ yield the same LRF curve: this is because both yield the exact same Markov chain, the only difference being that all fragment counts are multiplied by 3 in the former case.

Additionally, (6,3) with $t = 2$ that is "in between" (2,1) and (6,3) with $t = 1$. This result is quite intuitive: (6,3) with $t = 2$ yields less fragment dissemination than (6,3) with $t = 1$ but

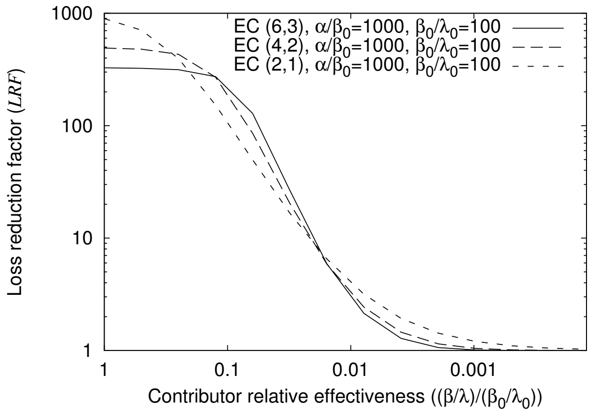


Figure 17. Data availability improvement as contributors become less effective for a given $\frac{\alpha}{\beta_0}$.

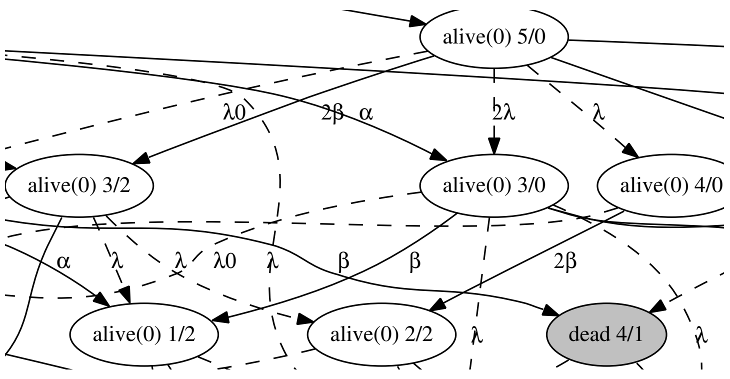


Figure 18. Excerpt from a Markov chain model for a (5,3) erasure code with $t = 2$ (two fragments transferred per contributor encounter). Dashed arrows denote failures.

more than (2,1) with $t = 1$ and, as seen in Section 3.4.3, more dissemination yields a lower *LRF*.

3.5. Related Work

Erasure codes have been widely used in (distributed) storage systems, but fewer works focus on a dependability assessment of erasure-code-based systems. In the area of Internet-based peer-to-peer backup and distributed storage, at least the Cooperative Internet Backup Scheme [Lillibridge *et al.* 2003], BAR-B [Aiyer *et al.* 2005], iDIBS [Morcos *et al.* 2006], OceanStore [Kubiatowicz *et al.* 2000] and InterMemory [Goldberg & Yianilos 1998] resort to erasure coding. The remainder of this section focuses on work related to the assessment of the dependability impact of erasure coding.

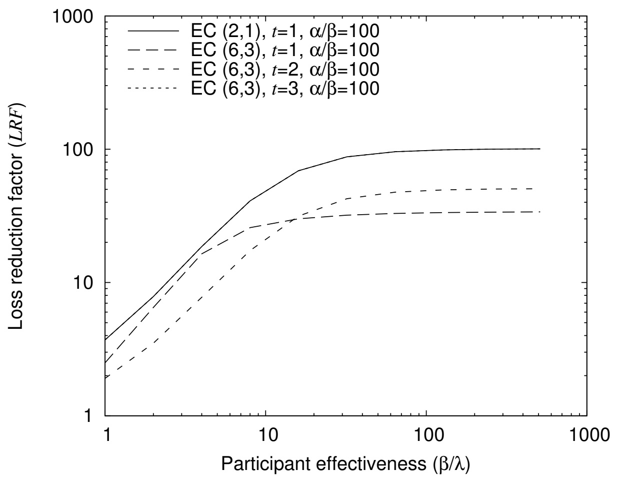


Figure 19. Data availability improvement with hybrid scattering strategies.

3.5.1. Erasure Codes in Distributed Storage Systems

In the context of MANETs, the Master report by Aspelund studies through simulation the *distance-to-backup* (in terms of number of hops) in the framework of a purely *ad hoc* cooperative backup service [Aspelund 2005] (see Section 2.4 for additional details). In this context, where absolutely no fixed infrastructure is considered accessible, one of the primary concerns is to minimize the distance between data owners and contributors. In the framework of the 7DS cooperative caching system for mobile devices, the authors studied how user mobility patterns affect the spread of information [Papadopouli & Schulzrinne 2000]. Again, we do not share this concern.

Several papers analyze data dependability in distributed and peer-to-peer storage systems. The authors of OceanStore conducted an analytical evaluation of the MTTF (mean time to failure) of a distributed, self-repairing storage system [Weatherspoon & Kubiatowicz 2002]. They conclude that erasure codes yield MTTF orders of magnitude higher than simple replication; however, their computations are based on probability distributions of hard disk failures, which may be quite different from that of individual untrusted peers on the Internet. Another paper about OceanStore makes a similar analysis: assuming that 10% out of all the nodes contributing to OceanStore are down at a given point in time, the fragmentation resulting from erasure coding increases reliability [Kubiatowicz *et al.* 2000]. Again, the assumptions that underlie this analysis are fairly optimistic and significantly different from ours: it is unlikely, in the scenarios we focus on, that 90% of all contributors storing data on behalf of a data owner may be reached by this data owner.

A similar comparison for peer-to-peer storage is proposed in [Vernois & Utard 2004], using a stochastic model. They conclude on the unsuitability of erasure codes in a peer-to-peer environment where peer availability is low. The major difference between these studies and what we present here is that the authors model a data block *repair* process that is inexistent in the context of a mostly-disconnected peer-to-peer backup system, notably because data owners cannot be made aware of contributor failures.

In [Lin *et al.* 2004], the authors analyze erasure code replication and compare the resulting data availability as a function of individual host availability (assuming each host stores exactly one fragment of the original data) and erasure code parameters (n,k). They identify a "switch point" between scenarios where erasure coding is preferable (from a data availability viewpoint) and scenarios where simple replication should be used. More precisely, they conclude that simple replication yields better data availability when host availability is low.

Our results comparing erasure codes and simple replication in terms of dependability are in agreement with those obtained on simpler models [Lin *et al.* 2004, Vernois & Utard 2004, Bhagwan *et al.* 2004]. We observe a switch point similar to that of [Lin *et al.* 2004]. For instance, in our model, whether erasure codes yield better data dependability than simple replication depends on $\frac{\alpha}{\beta}$ and $\frac{\beta}{\lambda}$ (see, e.g., Figure 13).

Building on a similar analysis, *TotalRecall* [Bhagwan *et al.* 2004], a peer-to-peer storage system, proposes mechanisms to automate availability management, which includes dynamic parameterization of erasure coding replication based on predicted host availability. However, the authors do not consider the use of erasure codes as a means to improve data confidentiality [Deswarte *et al.* 1991]. Additionally, the mobile environment we are addressing leads to a wider range of scenarios (and connectivity). A dynamic replication strategy for peer-to-peer cooperative data storage among untrusted nodes is also presented in [Ranganathan *et al.* 2002], though they do not consider the use of erasure codes.

Our partners within the MoSAIC project at IRISA have taken an approach that is complementary to ours. They provide formulæ that may be used for on-line assessment of the impact of the distribution of a single fragment on the overall data availability [Martin-Guillerez 2006]. Assuming the probability of successfully restoring individual fragments can be estimated, these formulæ may be used as a hint by the backup software to improve replica scheduling.

3.5.2. Erasure Codes in Delay-Tolerant Networks

Erasure codes have also been considered in the framework of delay-tolerant networks [Zhang 2006]. While flooding-based routing approaches provide the lowest delay and highest message delivery success rate at the cost of high bandwidth requirements, other routing strategies have been proposed that try to optimize bandwidth usage, delivery delay and success rate. In [Wang *et al.* 2005a], the authors discuss the use of erasure-coding based routing in DTNs whereby:

- the source messages are erasure-encoded;

- only the source of a message propagates fragments of the source message;
- relay nodes are allowed to send only to the destination.

This *two-hop* routing algorithm is similar to those we studied in this chapter. The authors compare through simulation this routing approach to a simple-replication approach according to several metrics: data success rate (probability that the message is delivered within a given amount of time), data latency, and routing overhead. They identify a tradeoff in terms of latency where: erasure coding yields lower worst case delay than simple replication, but provides lower data success rates for short deadlines.

The authors of [Jain *et al.* 2005] show that designing an optimal routing strategy is essentially a problem of optimizing *data allocation* among different possible paths. They note that optimizing both delay and delivery success rate is an NP-hard problem. Several allocation strategies with the same storage overhead are compared through simulation in different scenarios. Fragmentation (e.g., as a consequence of erasure coding) is shown to have a detrimental effect on message delivery success rate when individual path success probabilities are low (and all paths have the same success probability); conversely, fragmentation increases message delivery success probability when path success probabilities are higher. This is consistent with our results as well as those found in [Lin *et al.* 2004]. Additionally, simulation setups with heterogeneous path success probabilities allow the evaluation of routing strategies that use path probabilities as an input.

Similarly, Liao *et al.* studied the benefits of erasure coding in estimation-based routing for DTNs [Liao *et al.* 2006]. The idea is to have each source node estimate the *average contact frequency* (ACF) of each relay (or "contributor"); then, a source node distributes fragments of its message proportionally to the node's ACF (this is similar to one of the strategies evaluated in [Jain *et al.* 2005]). Simulation results show that estimation-based erasure coding routing (EBEC) performs better than estimation-based routing (EBRS) under the simulated scenario. However, only a narrow set of simulation parameters are considered, making it hard to compare the results to related work.

Finally, *network coding* was proposed as an alternative encoding mechanism for DTN routing [Widmer & Boudec 2005]. Using network coding, relay nodes may send out packets that are linear combinations of previously received information [Widmer & Boudec 2005]. Simulations show that network-coding-based probabilistic routing achieves higher message delivery rates than simple probabilistic routing for a given packet forwarding factor (i.e., bandwidth overhead). However, this strategy assumes multi-hop routing, which departs from our two-hop approach.

3.6. Summary

The contributions of this chapter are as follows:

- A model of the cooperative backup process based on Petri nets and Markov chains was proposed, along with a methodology for the dependability evaluation of the cooperative backup service.
- The dependability evaluation allowed the identification of *scenarios* where the cooperative backup approach is beneficial. Namely, we showed that the cooperative backup approach is beneficial (i.e., yields data dependability an order of magnitude higher than without MoSAIC) only when $\frac{\beta}{\lambda} > 2$ and $\frac{\alpha}{\beta} > 10$.
- We demonstrated that MoSAIC can decrease the *probability of data loss* by a factor up to the *ad hoc* to Internet connectivity ratio.
- The cooperative backup approach was shown to not improve data dependability when contributors are more than 100 times less effective than data owners. This result will have to be taken into account in the design of cooperation incentive and trust mechanisms (see Chapters 5 and 6).
- A comparison of simple replication and erasure codes and showed that erasure codes provide an advantage (dependability-wise) over simple replication only in narrow scenarios. Measurements of actual use cases are needed in order to see what real-world situations these scenarios map to.

Based on our results, several replication strategies can be envisioned. One possible strategy would be to maximize data dependability for a given user-specified storage overhead. Since in most scenarios little can be gained from using erasure codes, and since the consequence of a wrong decision would be detrimental to data dependability (e.g., choosing erasure coding in a scenario where simple replication would have been more beneficial), the best way to maximize data dependability is to always use simple replication.

Instead of focusing only on dependability, users may specify additional fragmentation to increase confidentiality [Deswarte *et al.* 1991]. Such a strategy could maximize fragmentation (i.e., by choosing a high k value) according to environmental parameters, while honoring a user-specified minimum dependability improvement factor. Environmental parameters, such as the data owner's failure rate, and Internet and *ad hoc* connectivity rates, as well as the effectiveness of encountered contributors, could be estimated based on past observations, perhaps augmented by user input.

Chapter 4. Storage Techniques

In this chapter, we focus on the mechanisms employed at the storage layer of our cooperative backup service. We identify fundamental requirements, investigate the various design options that satisfy them at this layer and discuss potential tradeoffs.

In Section 4.1, we detail the requirements of the cooperative backup service storage layer. Section 4.2 presents several design options for this layer based on the current literature and the particular needs that arise from the kind of devices we target. Section 4.3 gives an overview of our prototype storage layer implementation. Section 4.4 presents an evaluation of various storage layer algorithms using that prototype and discusses the necessary tradeoffs.

Part of the proposed design and experimental results shown in this chapter were published in [Courtès *et al.* 2006].

4.1. Requirements of the Storage Layer

In Chapter 2, we identified a number of high-level dependability goals for the envisioned cooperative backup service, along with lower-level requirements stemming from the wireless *ad hoc* backup process. Dependability goals included tackling threats to confidentiality and privacy, to integrity and authenticity, and to availability. Lower-level constraints included the ability to deal with data fragmentation and to provide energy- and storage-efficient backup techniques. In this section, we further details the requirements for the mechanisms employed at the storage layer.

Storage efficiency. Backing up data should be as efficient as possible. Data owners should neither ask contributors to store more data than necessary nor send excessive data over the wireless interface. Failing to do so will waste energy and result in inefficient utilization of the storage resources available in the node's vicinity. Inefficient storage may have a strong impact on energy consumption since (i) storage costs translate into transmission costs and (ii) energy consumption on mobile devices is dominated by wireless communication costs, which in turn increase as more data are transferred [Stemm *et al.* 1997]. *Compression techniques* are thus a key aspect of the storage layer on the data owner side.

Small data blocks. Both the occurrence of encounters of a peer within radio range and the lifetime of the resulting connections are unpredictable. Consequently, the backup application running on a data owner's device must be able to conveniently split the data to be backed up into small pieces to ensure that it can actually be transferred to contributors. Ideally, data blocks should be able to fit within the underlying link layer's maximum transmission unit or MTU (2304 octets for IEEE 802.11); larger payloads get fragmented into several packets, which introduces overhead at the MAC layer, and possibly at the transport layer too.

Backup atomicity. Unpredictability and the potentially short lifetime of connections, compounded with the possible use of differential compression to save storage resources, mean that it is impractical to store a set of files, or even one complete file, on a single peer. Indeed, it may even be undesirable to do so in order to protect data confidentiality [Deswarte *et al.* 1991]. Furthermore, it may be the case that files are modified before their previous version has been completely backed up.

The dissemination of data chunks as well as the coexistence of several versions of a file must not affect backup consistency as perceived by the end-user: a file should be either retrievable *and* correct, or unavailable. Likewise, the distributed store that consists of various contributors must remain in a "legal" state after new data are backed up onto it. This corresponds to the *atomicity* and *consistency* properties of the ACID properties commonly referred to in transactional database management systems.

Error detection. Accidental modifications of the data are assumed to be handled by the various lower-level software and hardware components involved, such as the communication protocol stack, the storage devices themselves, the operating system's file system implementation, etc. However, given that data owners are to hand their data to untrusted peers, the storage layer must provide mechanisms to ensure that *malicious* modifications to their data are detected with a high probability.

Encryption. Due to the lack of trust in contributors, data owners may wish to encrypt their data to ensure privacy. While there exist scenarios where there is sufficient trust among the participants such that encryption is not compulsory (e.g., several people in the same working group), encryption is a requirement in the general case.

Backup redundancy. Redundancy is the *raison d'être* of any data backup system, but when the system is based on cooperation, the backups themselves must be made redundant. First, the cooperative backup software must account for the fact that contributors may crash accidently. Second, contributor availability is unpredictable in a mobile environment without continuous Internet access. Third, contributors are not fully trusted and may behave maliciously. Indeed, the literature on Internet-based peer-to-peer backup systems describes a range of attacks against data availability, ranging from data retention (i.e., a contributor purposefully refuses to allow a data owner to retrieve its data) to selfishness (i.e., a participant refuses to spend energy and storage resources storing data on behalf of other nodes) [Lillibridge *et al.* 2003, Cox *et al.* 2002, Cox & Noble 2003]. All these uncertainties make redundancy even more critical in a cooperative backup service for mobile devices.

4.2. Design Options for the Storage Layer

In this section, we present design options able to satisfy each of the requirements identified for above.

4.2.1. Storage Efficiency

In wired distributed cooperative services, storage efficiency is often addressed by ensuring that a given content is only stored once. This property is known as *single-instance storage* [Bolosky *et al.* 2000]. It can be thought of as a form of compression among several data units. In a file system, where the "data unit" is the file, this means that a given content stored under different file names will be stored only once. On Unix-like systems, revision control and backup tools implement this property by using hard links [Lord 2005, Rubel 2005]. It may also be provided at a sub-file granularity, instead of at a whole file level, allowing reduction of unnecessary duplication with a finer-grain.

single-instance storage

Archival systems [Quinlan & Dorward 2002, You *et al.* 2005], peer-to-peer file sharing systems [Bennett *et al.* 2002], peer-to-peer backup systems [Cox *et al.* 2002, Landers *et al.* 2004], network file systems [Muthitacharoen *et al.* 2001], and remote synchronization tools [Tridgell & Mackerras 1996] have been demonstrated to benefit from single-instance storage, either by improving storage efficiency or reducing bandwidth. Interestingly, functional programming languages such as Scheme [Kelsey *et al.* 1998] have used a similar technique for immutable objects, referring to it as *interning* or *hash-consing* [Ershov 1958]: upon object (e.g., string, pair) construction, an object structurally equivalent to the one being created is looked up in a hash table and returned instead of being re-created. Again, this allows for memory savings.

Compression based on resemblance detection, that is, *differential compression*, or *delta encoding*, has been extensively studied [Hunt *et al.* 1996]. Proposals have been made to combine it with other compression techniques such as single-instance storage, even in situations that do not strictly relate to versioning [You & Karamanolis 2004, You *et al.* 2005, Kulkarni *et al.* 2004]. For each file to be stored, an exhaustive search over all stored files is performed to find the most similar file so that only the difference between these two files is stored. However, this technique is unsuitable for our environment since (i) it requires access to all the files already stored, (ii) it is CPU- and memory-intensive, and (iii) the resulting *delta chains* weaken data availability [You *et al.* 2005].

differential compression

Traditional *lossless compression* (i.e., *zip* variants), allows the elimination of duplication *within* single files. As such, it naturally complements inter-file and inter-version compression techniques [You *et al.* 2005]. Section 4.4 contains a discussion of the combination of both techniques in the framework of our proposed backup service. Lossless compressors usually yield better compression when operating on large input streams [Kulkarni *et al.* 2004] so compressing concatenated files rather than individual files improves storage efficiency [You *et al.* 2005]. However, we did not consider this approach suitable for mobile

lossless compression

device backup since it may be more efficient to backup only those files (or part of files) that have changed.

There exist a number of application-specific compression algorithms, such as the *lossless* algorithms used by the Free Lossless Audio Codec (FLAC), the PNG image format, and the XMill XML compressor [Liefke & Suciu 2000]. More generally, XMill introduced *type-driven compression*, where information about the type and structure of the data being compressed is leveraged to improve compression. There is also a plethora of *lossy* compression algorithms for audio samples, images, videos, etc. While using such application-specific algorithms might be beneficial in some cases, we have focused instead on generic lossless compression.

type-driven compression

4.2.2. Small Data Blocks

We now consider the options available to: (1) chop input streams into small blocks, and (2) create appropriate meta-data describing how those data blocks should be reassembled to produce the original stream.

4.2.2.1. Chopping Algorithms

As stated in Section 4.1, the size of blocks that are to be sent to contributors of the backup service has to be bounded, and preferably small, to match the nature of peer interactions in a mobile environment. There are several ways to cut input streams into blocks. Which algorithm is chosen has an impact on the improvement yielded by single-instance storage applied at the block level.

Splitting input streams into fixed-size blocks is a natural solution. When used in conjunction with a single-instance storage mechanism, it has been shown to improve the compression across files or across file versions [Quinlan & Dorward 2002]. Manber proposed an alternative *content-based stream chopping* algorithm [Manber 1994] that yields better duplication detection across files, a technique sometimes referred to as *content-defined blocks* [Kulkarni *et al.* 2004]. The algorithm determines block boundaries by computing Rabin fingerprints on a sliding window of the input streams. Thus, it only allows the specification of an *average* block size (assuming random input). Various applications such as archival systems [You *et al.* 2005], network file systems [Muthitacharoen *et al.* 2001] and backup systems [Cox *et al.* 2002] benefit from this algorithm. Section 4.4 provides a comparison of both algorithms.

content-based stream chopping

4.2.2.2. Stream Meta-Data

Placement of stream meta-data. Stream meta-data is information that describes which blocks comprise the stream and how they should be reassembled to produce the original stream. Such meta-data can either be embedded along with each data block or stored separately. The main evaluation criteria of a meta-data structure are read efficiency (e.g., algorithmic complexity of stream retrieval, number of accesses needed) and size (e.g., how the amount of meta-data grows compared to data).

We suggest a more flexible approach whereby stream meta-data (i.e., which blocks comprise a stream) is separated both from file meta-data (i.e., file name, permissions, etc.) and the file content. This has several advantages. First, it allows a data block to be referenced multiple times and hence allows for single-instance storage at the block level, as was already evidenced in earlier work on versioning such as CVFS [Soules *et al.* 2003]. Second, it promotes *separation of concerns.* For instance, file-level meta-data (e.g., file path, modification time, permissions) may change without having to modify the underlying data blocks, which is important in scenarios where propagating such updates would be next to impossible. Separating meta-data and data also leaves the possibility of applying the same "filters" (e.g., compression, encryption), or to use similar redundancy techniques for both data and meta-data blocks. This will be illustrated in Section 4.4. This approach is different from the one used in Hydra [Xu 2005], which separates meta-data from data and does not permit, for instance, the application of the same redundancy technique on meta-data as on data. However, it is comparable to OpenCM's approach [Shapiro & Vanderburgh 2002].

Indexing individual blocks. The separation of data and meta-data means that there must be a way for meta-data blocks to refer to data blocks: data blocks must be indexed or *named*[1]. The *block naming scheme* must fulfill several requirements. First, it must not be based on non-backed-up user state which would be lost during a crash. Most importantly, the block naming scheme must guarantee that *name clashes* among the blocks of a data owner cannot occur. In particular, block IDs must remain valid in time so that a given block ID is not wrongfully re-used when a device restarts the backup software after a crash. Given that data blocks will be disseminated among several peers and will ultimately migrate to their owner's repository, blocks IDs should remain valid in space, that is, they should be independent of contributor names. This property also allows for *pre-computation* of block IDs and meta-data blocks: stream chopping and indexing do not need to be done upon a contributor encounter, but can be performed *a priori*, once for all. This saves CPU time and energy, and allows data owners to immediately take advantage of a backup opportunity. A practical naming scheme widely used in the literature will be discussed in Section 4.2.4.

block naming scheme

Indexing sequences of blocks. Byte streams (file contents) can be thought of as sequences of blocks. Meta-data describing the list of blocks comprising a byte stream need to be produced and stored. In their simplest form, such meta-data are a vector of block IDs, or in other words, a byte stream. This means that this byte stream can in turn be indexed, recursively, until a meta-data byte stream is produced that fits the block size constraints, as illustrated in Figure 20. This approach yields the data structure shown in Figure 21, where leaves D_i represent data blocks, while I_i blocks are intermediary

[1] In the sequel we use the terms "block ID", "name", and "key" interchangeably.

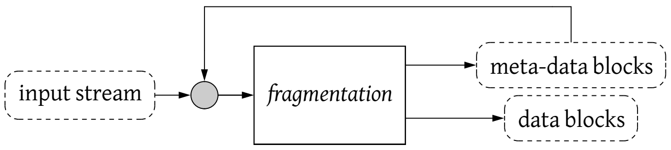


Figure 20. Data flow leading to the production of a series of blocks and meta-data blocks from an input data stream.

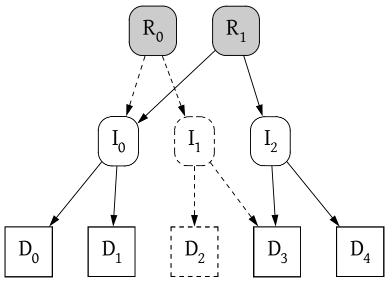


Figure 21. A tree structure for versioned stream meta-data.

meta-data blocks and roots R_i are root meta-data blocks. It is comparable to that used by Venti and GNUnet [Quinlan & Dorward 2002, Bennett *et al.* 2002].

Contributor interface. With such a design, contributors do not need to know about the actual implementation of block and stream indexing used by their clients, nor do they need to be aware of the data/meta-data distinction. All they need to do is to provide primitives of a keyed block storage:

- `put (key, data)` stores the data block `data` and associates it with `key`, a block ID chosen by the data owner according to some naming scheme;
- `get (key)` returns the data associated with `key`.

This simple interface suffices to implement, on the data owner side, byte stream indexing and retrieval. Also, it is suitable for an environment in which contributors and data owners are mutually suspicious because it places as little burden as possible on the contributor side. The same approach was adopted by Venti [Quinlan & Dorward 2002] and by many peer-to-peer systems [Bennett *et al.* 2002, Cox *et al.* 2002].

4.2.3. Backup Atomicity

Distributed and mobile file systems such as Coda [Lee *et al.* 1999] which support concurrent read-write access to data and do not have built-in support for revision control, differ significantly from backup systems. Namely, they are concerned about update propagation and reconciliation in the presence of concurrent updates. Not surprisingly, a read-write approach does not adapt well to the loosely connected scenarios we are targeting: data owners are not guaranteed to meet *every* contributor storing data on their behalf in a timely fashion when they need to update already backed-up data, which makes update propagation almost impossible. Additionally, it does not offer the desired atomicity requirement discussed in Section 4.1.

write once

The *write once* or *append only* semantics adopted by archival [Quinlan & Dorward 2002, Goldberg & Yianilos 1998], backup [Cox *et al.* 2002, Rubel 2005] and versioning systems [Santry *et al.* 1999, Lord 2005, Shapiro & Vanderburgh 2002] solve these problems. Data is always appended to the storage system, and never modified in place. This approach has long been acknowledged as preferable over in-place modification, as it is consistent with "good accounting practices" where entries are always added rather than modified [Gray 1981]. In practice, this is achieved by assigning each piece of data a unique identifier. Therefore, insertion of content (i.e., data blocks) into the storage space (be it a peer machine, a local file system or data repository) is atomic. Because data is only added, never modified, consistency is also guaranteed: insertion of a block cannot result in an inconsistent state of the storage space.

A potential concern with this approach is its cost in terms of storage resources. It has been argued, however, that the cost of storing subsequent revisions of whole sets of files can be very low, provided data that is unchanged across revisions is not duplicated, as described earlier [Santry *et al.* 1999, Gibson & Miller 1998, Quinlan & Dorward 2002]. In our case, once a contributor has finally transferred data to their owner's repository, it may reclaim the corresponding storage resources, which further limits the cost of this approach.

From an end-user viewpoint, being able to restore an old copy of a file is more valuable than being unable to restore the file at all. All these reasons make the append-only approach very suitable for the storage layer of our cooperative backup service.

4.2.4. Error Detection

Error-detecting codes can be computed either at the level of whole input streams or at the level of data blocks. They must then be part of, respectively, the stream meta-data, or the block meta-data. We argue the case for cryptographic hash functions as a means of providing the required error detection and as a block-level indexing scheme.

cryptographic hash functions

Cryptographic hash functions. The error-detecting code we use must be able to detect *malicious* modifications. This makes error-detecting codes designed to tolerate random, accidental faults inappropriate. We must instead use *cryptographic hash functions* , which are explicitly designed to detect tampering [NESSIE Consortium 2003a, NESSIE Consortium 2003b]. Such hash functions are usually characterized by three properties. First, cryptographic hash functions are *collision-resistant*, meaning that it should be hard to find two random input data that yield the same hash (i.e., there should be no method significantly more efficient than an exhaustive search). Second, they are *preimage-resistant* meaning that, given a hash, it should be computationally very expensive to find input data that yields it. Third, they are *second-preimage-resistant*, which means that given an input data block and its hash, it should be computationally difficult to find a second input data block yielding the same hash.

Along with integrity, *authenticity* of the data must also be guaranteed, otherwise a malicious contributor could deceive a data owner by producing fake data blocks along with valid cryptographic hashes[2]. Thus, digital signatures should be used to guarantee the authenticity of the data blocks. Fortunately, not all blocks need to be signed: signing a root meta-data block (as shown in Figure 21) is sufficient. This is similar to the approach taken by OpenCM [Shapiro & Vanderburgh 2002].

Content-based indexing. Collision-resistant hash functions have been assumed to meet the requirements of a data block naming scheme as defined in Section 4.2.2.2, and to be a tool allowing for efficient implementations of single-instance storage [Tridgell & Mackerras 1996, Cox *et al.* 2002, Muthitacharoen *et al.* 2001, You *et al.* 2005, Quinlan & Dorward 2002, Tolia *et al.* 2003, Landers *et al.* 2004]. In practice, these implementations assume that whenever two data blocks yield the same cryptographic hash value, their contents *are* identical. Given this assumption, implementation of a single-instance store is straightforward: a block only needs to be stored if its hash value was not found in the locally maintained block hash table.

In [Henson 2003], Henson argues that accidental collisions, although extremely rare, do have a slight negative impact on software reliability and yield silent errors. Given an n-bit hash output produced by a cryptographic hash function, the expected workload to generate a collision out of two *random* inputs is of the order of $2^{n/2}$ [NESSIE Consortium 2003a]. As an example, SHA-1, which produces 160-bit hashes, would require 2^{80} blocks to be generated on average before an accidental collision occurs. In our case, if a data owner is to store, say, 8 GiB of data in the form of 1 KiB blocks, we end up with 2^{43} blocks, which makes it unlikely that an accidental collision is hit.

We consider this to be reasonable in our application since it does not impede the tolerance of faults in any significant way. Also, Henson's fear of *malicious* collisions does not hold given the preimage-resistance property provided by the commonly-used hash functions[3]. Furthermore, it has been reported that most of Henson's examples in support of the idea that cryptographic hashes cannot be treated as unique identifiers for blocks of data are based on incorrect assumptions, misunderstandings, and questionable examples [Black 2006].

[2] Note, however, that while producing random data blocks and their hashes is easy, producing the corresponding meta-data blocks is next to impossible without knowing what particular meta-data schema is used by the data owner.

[3] The recent attacks found on SHA-1 by Wang et al. [Wang *et al.* 2005b] do not affect the preimage-resistance of this function.

Content-addressable storage (CAS) thus seems a viable option for our storage layer as it fulfills both the error-detection and data block naming requirements. In [Tolia *et al.* 2003], the authors assume a block ID space shared across all CAS users and providers. In our scenario, CAS providers (contributors) do not trust their clients (data owners) so they need either to enforce the block naming scheme (i.e., make sure that the ID of each block is indeed the hash value of its content), or to maintain a per-owner name space.

Content-addressable storage

Combining the tree meta-data structure as shown in 21 with content-addressable storage through cryptographic hash functions effectively yields a so-called *Merkle hash tree* [Merkle 1980]. This data structure has the interesting property that if the tree root has been authenticated, then intermediate nodes and leaves (data blocks) can be considered genuine, too, as already mentioned above.

Merkle hash tree

4.2.5. Encryption

Data encryption may be performed either at the level of individual blocks, or at the level of input streams. Encrypting the input stream *before* it is split into smaller blocks breaks the single-instance storage property at the level of individual blocks. This is because encryption aims to ensure that the encrypted output of two similar input streams will not be correlated.

Leaving input streams unencrypted and encrypting individual blocks yielded by the chopping algorithm does not have this disadvantage. More precisely, it preserves single-instance storage at the level of blocks at least *locally*, i.e., on the data owner side. If asymmetric ciphering algorithms are used, the single-instance storage property is no longer ensured *across* peers, since each peer encrypts data with its own private key. However, we do not consider this to be a major drawback for the majority of scenarios considered, since little or no data are common to several participants. Moreover, solutions to this problem exist, notably *convergent encryption* [Cox *et al.* 2002].

convergent encryption

In Chapter 5, we elaborate on practical encryption schemes and discuss their integration with other security mechanisms.

4.2.6. Backup Redundancy

Replication strategies. Redundancy management in the context of our collaborative backup service for mobile devices introduces a number of new challenges. Peer-to-peer file sharing systems are not a good source of inspiration in this respect given that they rely on redundancy primarily as a means of reducing access time to popular content [Ranganathan *et al.* 2002].

In Chapter 3, we considered the use of *erasure codes* as a means to introduce data redundancy. Should erasure codes be used, they could be applied either to the input data stream, or at the block-level, regardless of whether blocks are data or meta-data blocks. When applied at the block-level, erasure coding could be handled through a slight block naming change: erasure-coding a block yields n coded blocks, which must each have a different name, but it should be easy to derive their name from the original block name. The names of the erasure-coded blocks could be derived from that of the original block, e.g., by appending a sequence number. In this way, the recovery process could easily query erasure-coded blocks by just appending a correct prefix to the original block name. For instance, assuming a block named a yields coded blocks a_0, a_1 and a_2, out of which any 2 coded blocks suffice to recover the original block, then the recovery process, when asked for a, would just try to fetch a_0, then a_1, etc., until 2 of these coded blocks have been fetched. However, we concluded in Chapter 3 that few scenarios would benefit from such uses of erasure coding, since they increase fragmentation, which in turn slightly increases the probability of data loss in most scenarios (Section 3.4.3).

Nevertheless, erasure coding may be beneficial in terms of data dependability when applied to the entire input data stream, in lieu of one of the chopping algorithms discussed earlier, as mentioned in [Jain *et al.* 2005]. Indeed, several strategies with the same storage overhead and the same number of blocks (i.e., strategies that do *not* increase fragmentation) can be found by changing both the chopping and erasure coding parameters. Figure 22 shows the possible strategies with a storage overhead of 2 and 16 blocks to distribute[4]: the first strategy does not chop its input stream but directly applies a (16,8) erasure code, the second strategy chops its input stream into 2 blocks and applies an (8,4) erasure code to *each* block, and so on.

For each strategy shown in Figure 22, only 8 blocks need to be fetched to recover the original data. However, with the first strategy, *any* 8 blocks out of 16 must be recovered, whereas with the last strategy, exactly one copy of each of the 8 input blocks must be fetched. This makes a significant difference in the number of possibilities by which the input data can be recovered, as shown in the third column of Figure 22. This would make the first strategy the best one from a dependability viewpoint. The key insight here is that erasure coding provides better dependability when applied at the level of "logical data units" (e.g., whole files), rather than on a subset thereof (e.g., fraction of a file).

This analysis differs from our comparison of erasure codes and simple replication in Section 3.4.3 in that (i) the comparison did not consider data chopping (which is equivalent to saying that the number of input blocks was always 1), and (ii) only the storage overhead, i.e., $\frac{n}{k}$, was fixed. Here, we further constrain our comparison by fixing the total number of blocks to distribute, i.e., n times the number of input blocks. Our Petri net model (see Chapter 3) could be extended to handle chopping and allow for a more detailed comparison.

Erasure coding thus appears to be a valid option *both* as an input stream chopping and redundancy mechanism. However, doing so would require erasure coding algorithms that provide the necessary flexibility on the n and k parameters. The meta-data may itself be

[4] For simplicity, we do not consider meta-data blocks here.

Number of Input Blocks	Erasure Code	Recovery Possibilities
1	(16,8)	$\binom{16}{8}^1 = 12870$
2	(8,4)	$\binom{8}{4}^2 = 4900$
4	(4,2)	$\binom{4}{2}^4 = 1296$
8	(2,1)	$\binom{2}{1}^8 = 256$

Figure 22. Comparison of erasure coding and regular chopping and redundancy mechanisms.

erasure-coded in a similar way, until the size of an individual meta-data fragment is small enough, as illustrated in Figure 20.

On the other hand, using erasure coding on input data streams precludes a number of storage optimizations previously discussed, such as sharing of data common to several revisions of a file. For immutable input streams (e.g., audio/video files), this is not a problem. It might degrade storage efficiency for, e.g., versioned textual files. Which chopping and redundancy strategy is the best to each type of file is an open issue.

Replica scheduling and dissemination. As stated in Section 4.1, it is plausible that a file will be only partly backed up when a newer version of this file enters the backup creation pipeline. One could argue that the replica scheduler should finish distributing the data blocks from the old version that it started to distribute before distributing those of the new version. This policy would guarantee, at least, availability of the old version of the file. On the other hand, in certain scenarios, users might want to favor freshness over availability, i.e., they might request that newer blocks are scheduled first for replication (Section 6.3.1 further discusses such issues).

This clearly illustrates that a wide range of *replica scheduling and dissemination policies and corresponding algorithms* can be defended depending on the scenario considered. At the core of a given replica scheduling and dissemination algorithm is a *dispersal function* that decides on a level of dispersal for any given data block. The level of dispersal can evolve *dynamically* to account for several changing factors. In FlashBack [Loo *et al.* 2003], the authors identify a number of important factors that they use to define a *device utility function*. Those factors include *locality* (i.e., the likelihood of encountering a given device again later) as well as *power and storage resources* of the device.

In addition to those factors, our backup software needs to account for the current level of trust in the contributor at hand. If a data owner fully trusts a contributor, e.g., because it has proven to be well-behaved over a given period of time, the data owner may choose to store complete replicas (i.e., mirrors) on this contributor.

4.3. Implementation Overview

libchop

We have implemented a prototype of the storage layer discussed above. As this layer is performance-critical, we implemented it in C. The resulting library, *libchop*, consists of 10 k source lines of code. The library was designed in order to be flexible so that different techniques could be combined and evaluated. To that end, the library itself consists of a few well-defined interfaces.

The following section presents the data flow during the storage process through *libchop*'s storage pipeline. We then discuss the data retrieval process as well as support for distributed storage. Note that additional information is available in Appendix .

4.3.1. Storage Pipeline

Figure 23 shows a UML class diagram of the main *libchop* components, while Figure 24 informally illustrates the data flow through *libchop*'s main components during the storage process. Each box represents a *programming interface* of the library. Several implementations are available for each interface, as shown in Figure 23, which allowed us to conduct the experiments to be described in Section 4.4. The library itself is not concerned with the backup of file system-related meta-data such as file paths, permissions, etc. Implementing this is left to higher-level layers akin to OpenCM's schemas [Shapiro & Vanderburgh 2002], which will be discussed in Chapter 6.

Input data streams such as file contents are represented by the `stream` interface. The `chopper` interface defines a layer over input streams from which entire, successive blocks can be fetched. Three classes implement this interface: one that yields fixed-size blocks, one that yields variable-size blocks according to Manber's algorithm [Manber 1994], and another that returns the whole input stream as a single block (i.e., it does not chop the input stream). Note that we did not implement erasure coding as suggested in Section 4.2.6.

The `block_indexer` interface provides a method that, given a data block, stores it into some block store and returns an *index*. An index is an opaque object that can be serialized either in a compact binary form or in a printable ASCII form. That index objects are opaque guarantees that users of a block indexer implementation can transparently adapt to any other implementation. There are currently two implementations of the `block_indexer` interfaces that do *not* guarantee single-instance storage:

- the stateless `uuid` block indexer, which uses globally unique block identifiers (per RFC 4122 specifications), using `libuuid`;

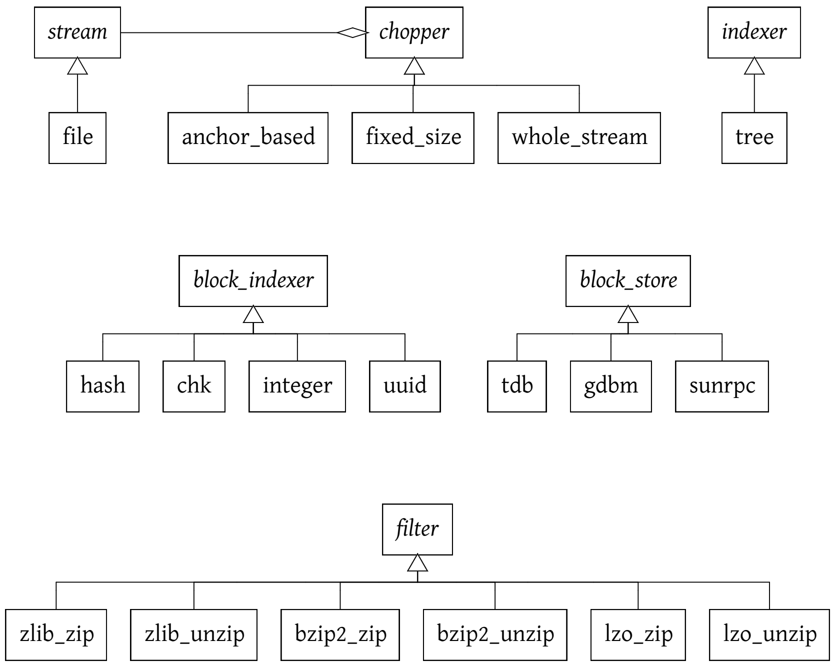


Figure 23. UML class diagram of *libchop*.

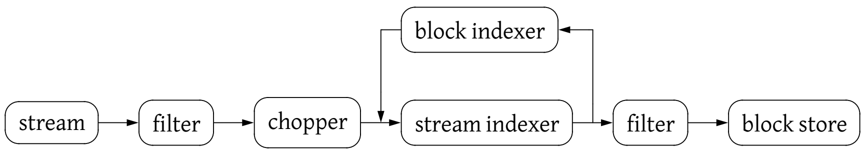


Figure 24. Data flow in the *libchop* backup creation pipeline.

- the `integer` block indexer, which uses 32-bit integers as block identifiers, starting from zero and incrementing each a block is indexed.

In addition, two stateless block indexers that guarantee single-storage instance are available:

- the `hash` block indexer, which uses the cryptographic hash of a block as its identifier, using SHA-1 or other algorithms;
- the `chk` block indexer, which implements *content-hash keys* (CHK) as used in Freenet [Clarke *et al.* 2001] and GNUnet [Bennett *et al.* 2002] (see Section 5.2).

content-hash keys

Unless otherwise stated, the experiments presented in Section 4.4 use either the `integer` or the`hash` block indexer.

The `stream_indexer` interface provides a method that iterates over the blocks yielded by the given chopper, indexes them, produces corresponding meta-data blocks, and stores them into a block store. There is currently only one implementation of this interface, namely a "tree indexer" that produces meta-data in a form similar to that shown in Figure 21. To date, we have not thought of other worthwhile implementations of this interface.

Finally, The `block_store` interface mainly consists of the `put` and `get` methods described in Section 4.2.2.2. Available implementations of this interface allow blocks to be store either in an on-disk database (such as TDB [Tridgell *et al.* 1999]) or over the network to a server that implements the corresponding RPC interface (see Section 4.3.3).

libchop also defines a `filter` interface. Filters may conveniently be reused in different places, for instance between a file-based input stream and a chopper, or between a stream indexer and a block store. They are used to implement compression and decompression filters using the *zlib* [Deutsch & Gailly 1996, Deutsch 1996], *bzip2* [Seward 2007] and *LZO* [Oberhumer 2005] libraries. Bzip2 is very storage-efficient but also very memory- and CPU-intensive. Conversely, LZO trades storage efficiency for compression and decompression speed. Encryption and decryption filters could also be imagined, although none is currently implemented.

4.3.2. Data Retrieval Process

Some of the interfaces presented above have a tightly coupled "dual" interface providing functionality for the backup retrieval pipeline. For instance, the interface corresponding to `block_indexer` is `block_fetcher`. Conversely, the `chopper` interface does not have a dual interface. This is because the blocks yielded by the `chopper` interface are assumed to be contiguous, and thus only need to be concatenated upon retrieval. More information on the actual retrieval process may be found in Appendix

4.3.3. Support for Distributed Storage

In the proposed cooperative backup service, chopping and indexing are performed on the data owner side, while the block store itself is realized by contributors. Thus, from an architectural viewpoint, contributors naturally fit as an implementation of the `block_store` interface.

Concretely, a `sunrpc_block_store` class implements the `put` and `get` method of the `block_store` interface using the Sun/ONC Remote Procedure Call (RPC) mechanisms [Srinivasan 1995]. This suffices to allow data storage to a remote site. MoSAIC contributors only need to implement this simple RPC interface.

Chapter 5 will present security extensions to this simple protocol, whereby ONC RPC are transported on a TLS-authenticated channel. These facilities are also implemented as part of the `sunrpc_block_store` class.

4.4. Experimental Evaluation

This section presents our prototype implementation of the storage layer of the envisaged backup system, as well as a preliminary evaluation of key aspects.

Our implementation has allowed us to evaluate more precisely some of the tradeoffs outlined in Section 4.2. After describing the methodology and workloads that were used, we will comment the results obtained.

4.4.1. Methodology

The purpose of our evaluation is to compare the various compression techniques described earlier in order to better understand the tradeoffs that must be made. We measured the storage efficiency and computational cost of each method, both of which are critical criteria for resource-constrained devices. The measures were performed on a 500 MHz G4 Macintosh running GNU/Linux (running them on, say, an ARM-based mobile device would have resulted in lower throughputs; however, since we are interested in *comparing* throughputs, this would not make any significant difference).

We chose several workloads and compared the results obtained using different configurations. These file sets, described in Section 4.4.2, qualify as *semi-synthetic* workloads because they are actual workloads, although they were not taken from a real mobile device. The motivation for this choice was to purposefully target specific file *classes*. The idea is that the results should remain valid for any file of these classes.

Configurations. Figure 25 shows the storage configurations we have used in our experiments. For each configuration, it indicates whether single-instance storage was provided, which chopping algorithm was used and what the expected block size was, as well as whether the input stream or output blocks were compressed using a lossless stream compression algorithm. We instantiated each configuration with each of the three lossless compressing filters available, namely *zlib*, *bzip2* and LZO (see Section 4.3.1). Our intent is not to evaluate the outcome of each algorithm independently, but rather that of whole configurations. Thus, instead of experimenting with every possible combination, we chose to retain only those that (i) made sense from an algorithmic viewpoint and (ii) were helpful in understanding the tradeoffs at hand.

For all configurations, the only stream indexer used is a “tree indexer”. We used an on-disk block store that uses TDB as the underlying database [Tridgell *et al.* 1999]—in other words, data blocks were *not* transferred over the network, only stored locally. For each file set, we started with a new, empty database. Since the computational cost of the insertion of a new entry is a function of the number of entries already present in the database, this will have an impact on the execution time for large file sets. However, this does not preclude comparing the execution times for a given file set with different configurations.

Configurations A_1 and A_2 serve as baselines for the overall compression ratio and computational cost. Comparing them is also helpful in determining the computational cost

Config.	Single Instance?	Chopping Algo.	Expected Block Size	Input Zipped?	Blocks Zipped?
A_1	no	—	—	yes	—
A_2	yes	—	—	yes	—
B_1	yes	Manber's	1024 B	no	no
B_2	yes	Manber's	1024 B	no	yes
B_3	yes	fixed-size	1024 B	no	yes
C	yes	fixed-size	1024 B	yes	no

Figure 25. Description of the configurations experimented.

due to single-instance storage alone.Subsequent configurations all chop input streams into small blocks whose size fits our requirements (1 KiB, which should yield packets slightly smaller than IEEE 802.11's MTU); they all implement single-instance storage of the blocks produced. However, we also study the impact of varying block sizes in a later experiment, using variants of configuration B_1.

Common octet sequences are unlikely to be found *within* a *zlib*-compressed stream, by definition. Hence, zipping the input precludes advantages to be gained by block-level single-instance storage afterwards. Thus, we did not include a configuration where a zipped input stream would then be passed to a chopper implementing Manber's algorithm.

The *B* configurations favor sub-file single-instance storage by not compressing the input before chopping it. B_2 improves over B_1 by adding the benefits of compression at the block-level. Conversely, configuration *C* favors traditional lossless compression over sub-file single-instance storage since it applies lossless compression to the input stream.

Our implementation of Manber's algorithm uses a sliding window of 48 B which was reported to provide good results [Muthitacharoen *et al.* 2001]. All configurations but A_1 use single-instance storage, realized using the *libchop* `hash` block indexer that uses SHA-1 hashes as unique block identifiers. For A_1, an `integer` block indexer, which systematically provides unique IDs, was used.

Unlike *zlib* and *bzip2*, the library that implements LZO does not manage input data buffering internally. Lossless compressors can usually achieve better compression at the cost of using more memory at run-time. In the following experiments, the LZO compression filter always used a 16 KiB input buffer, which is what *zlib* does by default. For LZO compression, the LZO 1X algorithm was used; for *zlib*, the default compression level was used; for *bzip2* the default "work factor" was used, along with one 100 KiB block for input[5].

[5] This is the minimum value allowed as the `blockSize100k` parameter of `BZ2_bzCompressInit ()`. This is

The chosen configurations and file sets are quite similar to those described in [You *et al.* 2005, Kulkarni *et al.* 2004, You & Karamanolis 2004], except that, as explained in Section 4.2.1, we do not evaluate the storage efficiency of the differential compression technique proposed therein.

4.4.2. Workloads

In Figure 26, the first file set contains 10 successive versions of the source code of the Lout document formatting system, i.e., low-density, textual input (C and Lout code), spread across a number of small files. Of course, this type of data is not typical of mobile devices like PDAs and cell phones. Nevertheless, the results obtained with this workload should be similar to those obtained with widely-used textual data format such as XML. The second file set shown in Figure 26 consists of 17 Ogg Vorbis files, a high-density binary format[6], typical of the kind of data that may be found on devices equipped with sampling peripherals (e.g., audio recorders, cameras). The third file set consists of a single, large file: a mailbox in the Unix mbox format which is an append-only textual format. Such data are likely to be found in a similar form on communicating devices.

4.4.3. Results

Figures 27, 28 and 29 show the results obtained for each file set. Each figure contains three charts presenting the results of all the configurations of Figure 25, instantiated with each of the three compression filters. Each plot shows the *throughput* (in octets per time unit[7], where both system and user time are accounted for) versus the corresponding data *compression ratio.* The compression ratio is defined as the ratio of the size of the resulting blocks, *including* meta-data (sequences of block indices), to the size of the input data. The throughput is defined as the experiment execution time over the input data size in octets. When measuring the execution time, each experiment was run 10 times; horizontal lines on the charts denote the standard deviation computed over the set of execution time measurements. Note that configuration B_1 remains at the same position on all three plots for a given file set since it does not use any lossless compressor.

Impact of the data type. Not suprisingly, the set of Ogg Vorbis files defeats all the compression techniques. Most configurations incur a slight storage overhead due to the amount of meta-data generated. This input data is a pathological case for all lossless compressors, both from a storage-efficiency and data throughput perspective. Nevertheless, LZO displays much better throughput and comparable storage efficiency in such pathological cases.

more memory than what the two other compressors use.

[6] Ogg Vorbis is a lossy audio compression format. See *http://xiph.org/* for details.

[7] The "time unit" is platform-specific, as returned by the POSIX.1 `times ()` function.

Name	Size	Files	Avg. Size
Lout (versions 3.20 to 3.29)	76 MiB	5853	13 KiB
Ogg Vorbis files	69 MiB	17	4 MiB
mbox-formatted mailbox	7 MiB	1	7 MiB

Figure 26. File sets.

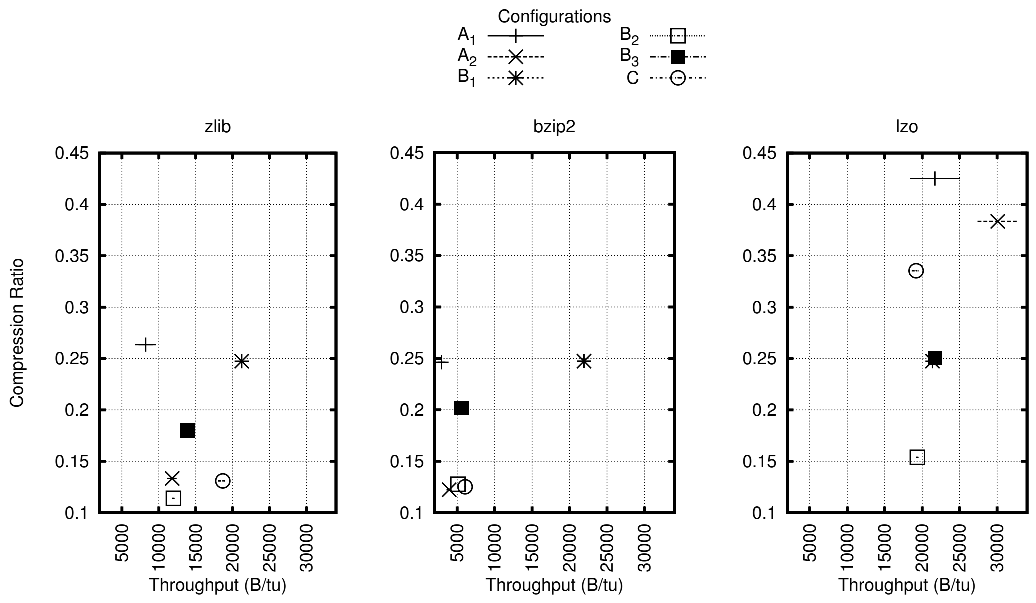


Figure 27. Storage efficiency and throughput of several configurations and lossless compressors for the Lout file set.

Impact of single-instance storage. Comparing the results obtained for A_1 and A_2 shows benefits only in the case of the successive source code distributions (Figure 27), where it halves the amount of data stored when *zlib* and *bzip2* input compression is used. This is due to the fact that successive versions of the software have a lot of files in common. Furthermore, these experiments show that, even when no compression advantage is obtained, single-instance storage implemented using cryptographic hashes does not significantly degrade throughput; in some cases, e.g., on Figure 27, it even slightly improves throughput because fewer writes to the block store are needed. Consequently, we chose to use single-instance storage in all configurations.

Surprisingly, this behavior is not observed when LZO is used: this is because builds of the LZO library, by default, are non-deterministic[8]; non-determinism also explains the relatively high execution time standard deviations observed with LZO input stream

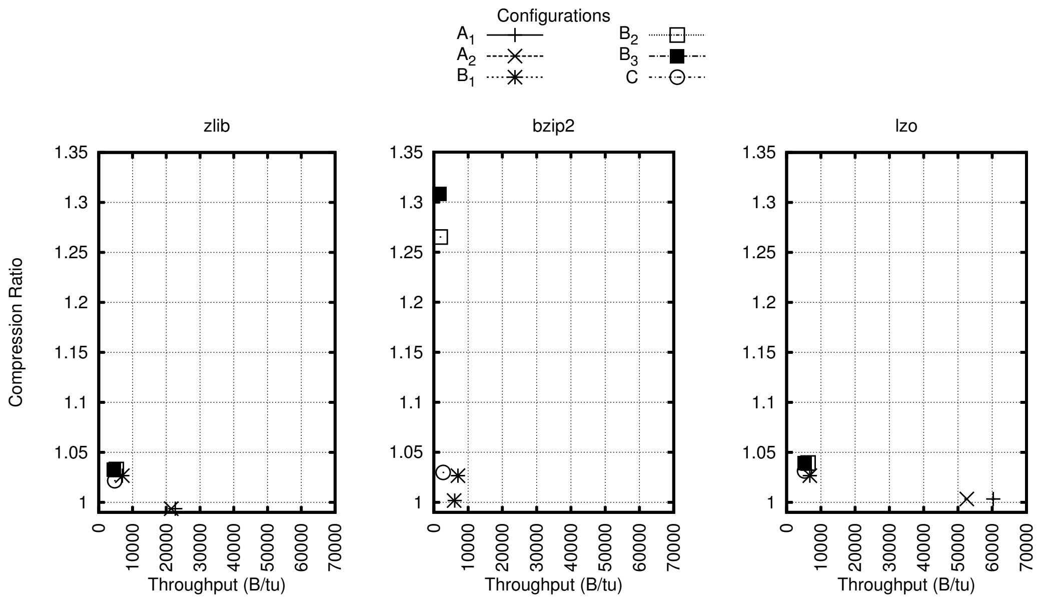


Figure 28. Storage efficiency and throughput of several configurations and lossless compressors for the Ogg Vorbis file set.

compression. We did not try compiling the LZO library with the appropriate flag that makes it deterministic.

As expected, single-instance storage applied at the block-level is mainly beneficial for the Lout file set where it achieves noticeable inter-version compression, comparable with that produced with *zlib* in A_1. The best compression ratio overall is obtained with B_2 where individual blocks are *zlib*-compressed. It is comparable to the compression obtained with *C*, though, and only slightly better in the Lout case (11 % vs. 13 %). The results in [You *et al.* 2005] are slightly more optimistic regarding the storage efficiency of a configuration similar to B_2 with *zlib* compression, which may be due to the use a smaller block size (512 B) and a larger file set.

Impact of the block size. In addition to the end-to-end measurements presented earlier, it seemed worthwhile to focus on the impact of block sizes on storage efficiency. When Manber's algorithm is used, using smaller block sizes can help detect more redundancy. Conversely, it also yields more data blocks, hence more meta-data. Therefore, decreasing the expected block size may be beneficial for file sets that expose a lot of redundancy, but may be counter-productive for other file sets.

[8] According to comments by the author in the source code of version 1.08 of the library, having non-deterministic behavior was motivated by performance reasons. Specifically, the author notes that non-determinism may be helpful « when the block size is very small (e.g., 8kB) or the dictionary is big, because then the initialization of the dictionary becomes a relevant magnitude for compression speed. »

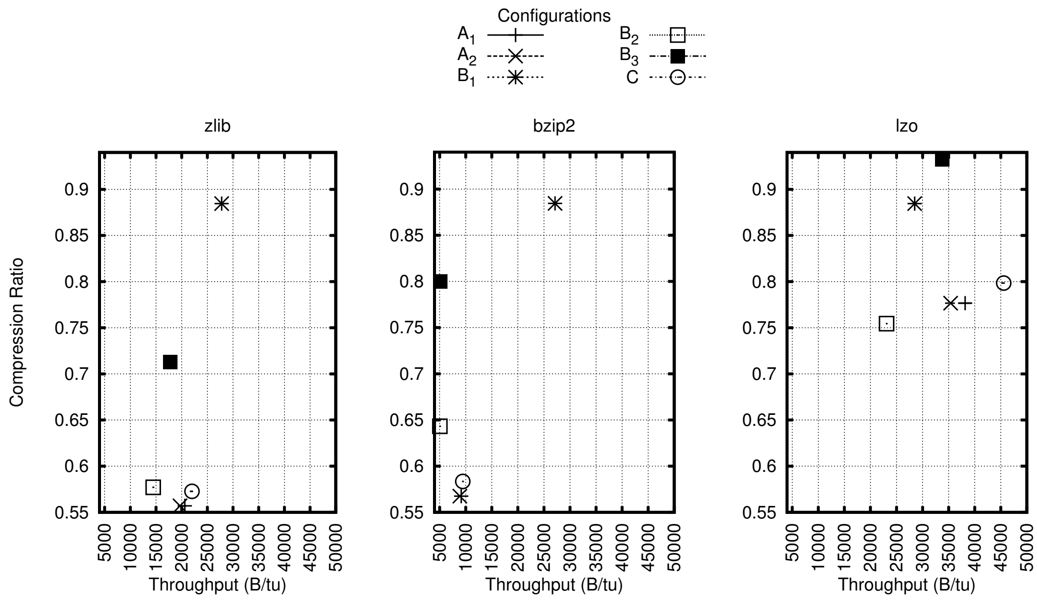


Figure 29. Storage efficiency and throughput of several configurations and lossless compressors for the mailbox file.

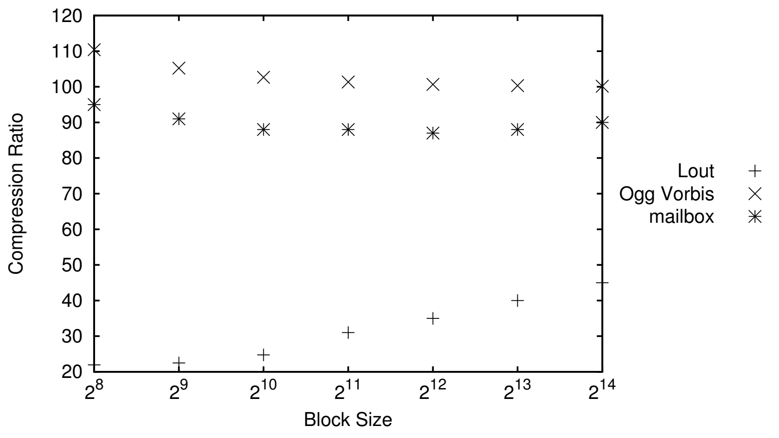


Figure 30. Compression level with single-instance storage combined with Manber's algorithm vs. block size.

Figure 30 shows the compression ratio obtained for configurations similar to B_1 but with different block sizes. The outcome is that the Lout file set benefits from smaller block sizes, even with blocks as small as 256 octets. These results are comparable to those obtained by the authors of LBFS for small edits [Muthitacharoen *et al.* 2001]. Conversely, the mailbox file sets reaches an optimum with block sizes around 4 KiB, while the Ogg Vorbis file set is stored more efficiently using the largest investigated block size.

With smaller block sizes, storage efficiency for the mailbox and Ogg Vorbis files quickly starts decreasing because meta-data overhead outweighs the efficiency gain; at some point, the resulting data size is even larger than the input data size. Nevertheless, it is worth noting that the relative size of meta-data with 1 KiB blocks and using a `hash` block indexer with 20-octet SHA-1 hashes (as used in Figure 30 and for B_1) is negligible (less than 3 % of the total output).

Computational cost. If we take into account throughput as well, we see that *bzip2* is almost always prohibitively expensive compared to *zlib* and LZO. For the Lout and mailbox file sets, the two best candidate configurations are C with *zlib* compression, or B_2 with LZO compression. The latter is slightly less CPU-intensive and also slightly less storage-efficient than the former.

Additional conclusions can be drawn with respect to throughput. First, the cost of *zlib*-based compression appears to be reasonable, particularly when performed on the input stream rather than on individual blocks, as evidenced, e.g., by B_3 and C. Second, the input data type has a noticeable impact on the throughput. In particular, applying lossless compression is more CPU-intensive for the Ogg Vorbis files than for low-entropy data. The results are even more disastrous with configuration B_2 where lossless compression is applied at the block level. Therefore, it would be worthwhile to disable lossless compression for compressed data types.

4.5. Related Work

The authors of the *Deep Store* archival system and the authors of *Redundancy Elimination at the Block Level* (REBL) conducted experiments similar to ours that aim at measuring the storage efficiency of various storage techniques, using data sets representative or large archival stores [You *et al.* 2005, You & Karamanolis 2004, Kulkarni *et al.* 2004]. As far as single-instance storage and content-defined file chopping is concerned, both studies report storage-efficiency results comparable to ours. However, only [Kulkarni *et al.* 2004] measured execution time. Both studies evaluated differential compression, which we rejected for reasons outlined earlier (see Section 4.2.1). Instead, our work focuses on configurations and data sets more suitable to the peer-to-peer mobile environment targeted. It also augments these studies with a comparison of storage configurations involving different lossless compression algorithms.

For the experiments we made, we made the rough assumption that CPU energy costs are proportional to the time spent encoding the data, and that wireless transmission costs are proportional to the amount of data to be transmitted. The energy costs associated with the various design options could be assessed more precisely, especially those related to the wireless transmission of backup data between nodes. In [Barr & Asanovic 2006], the authors instrumented a Wi-Fi-capable hardware platform to examine its power consumption. They concluded that CPU energy consumption is, indeed, roughly proportional to the execution time. They also note that, although Wi-Fi interfaces are very power-hungry, the energy spent in compressing a file approaches or outweighs the energy needed to transmit it for most compressors, LZO and *zlib* with the lowest compression level being notable exceptions. Among the reasons invoked is the fact that the most storage-efficient compressors, such as *bzip2*, generate a lot of memory traffic beside being CPU-intensive, and that memory traffic is also very energy-consuming.

This complex tradeoff is fundamental for mobile distributed applications. We believe that one way to better understand it would be by (i) modeling the power consumption of the main hardware components of a mobile devices (wireless network interface, CPU, memory) and (ii) using those models to estimate the actual energy consumption of various configurations. As far as CPU and memory are concerned, tools such as Valgrind's Cachegrind tool [Nethercote & Seward 2007] already allow the emulation of hardware platforms at a fine-grain level, and can report information such as memory accesses, L1 and L2 cache misses, etc. For instance, it has been successfully used to estimate the energy cost associated with memory paging policies [Park *et al.* 2004]. Nevertheless, besides being hard to set up, such an approach suffers from being very tied to specific hardware and software implementations.

Compact encoding of *structured* application data seems like a promising approach. When operating on arbitrary files or octet streams, our backup software does not have the opportunity to take advantage of the data structure they represent. XMill [Liefke & Suciu 2000], an XML compressor, demonstrated that using information about the structure and type of the data being compressed can yield significant improvements in terms of storage-efficiency, by applying type-specific encoding and compression algorithms. Many high-level programming languages provide run-time type information which could be leveraged to store an arbitrary object graph in a compact or compressed form. In fact, the run-time systems of those languages have already incorporated *ad hoc* type-specific encoding rules for some time [Bobrow & Clark 1979]; the same is true, of course, of widely used binary serialization formats such as XDR [Eisler 2006].

Future work on the optimization of the storage layer concerns several aspects. First, beside compression techniques, other parts of the storage layer need to be assessed, in particular *encryption* techniques. In particular, the CPU cost of public-key encryption ought to be compared with that of symmetric encryption (e.g., based on content-hash keys as in [Clarke *et al.* 2001, Bennett *et al.* 2002]). Again, comparisons could be made by applying these techniques either at the input stream level or at the block level through *libchop* "filters". Fortunately, our storage framework is well-suited to such additions.

Finally, it seems likely that no single configuration of the backup service will be appropriate for all situations. Thus, dynamic adaptation of the service to suit different contexts could be investigated. Storage configurations could be chosen either based on user input, or based on the (alleged) type of the files being encoded. In particular, this would allow compressed file types to be handled gracefully, with no performance loss, while aggressively combining suitable compression techniques for versioned textual data, for instance.

4.6. Summary

In this chapter, we studied the design and implementation of a suitable storage layer for the envisioned cooperative backup service. Key points included the following:

- We identified six essential requirements for the storage layer of such a service, namely: (i) storage efficiency; (ii) small data blocks; (iii) backup atomicity; (iv) error detection; (v) encryption; (vi) backup redundancy.
- Various design options meeting these requirements have been reviewed.
- Erasure coding was suggested as a way to handle both input stream chopping and redundancy.
- We presented our flexible storage layer implementation, named *libchop*, has been presented.
- An experimental evaluation of several combinations of storage techniques has been carried out using our implementation.

The contribution of the experimental evaluation presented in this chapter can be summarized as follows:

- We assessed and compared different storage techniques, both in terms of storage efficiency and computational cost.
- We observed that, using 20-octet hashes to designate 1 KiB blocks, meta-data size is negligible.
- We conclude that the most suitable combination for our purposes combines the use of lossless input compression with fixed-size chopping and single-instance storage.
- Other techniques were rejected for providing little storage efficiency improvement compared to their CPU cost.

We believe that these results could be useful in other storage areas with varying criteria, ranging from archival storage to revision control systems. In addition, our software framework is flexible enough to allow similar experimentations to be made with little effort.

Chapter 5. Secure Cooperation

A number of threats to data confidentiality, integrity, and availability, arise in our cooperative backup service, as seen in Section 2.2. Until now, we have focused primarily on threats to data availability that arise as a consequence of *accidental* failures. Here, we consider threats to both data and service availability stemming from *malicious* uses of the cooperative backup service. Indeed, since anyone can participate in our cooperative backup service, users of the service are *mutually distrustful* and both the service and its users may be subject to attacks.

Section 5.1 summarizes the dependability goals defined in Section 2.2, highlighting those that have already been targeted by previous chapters and those still to be addressed. Section 5.2 provides an overview of the storage layer presented in Chapter 4, identifying the security concerns that it addresses and those it does not address. Section 5.3 proposes self-organized, policy-neutral security mechanisms and shows (i) how they fulfill some of our requirements and (ii) how they can be used as building blocks for various cooperation policies. Section 5.4 deals with implementation considerations. Section 5.5 summarizes related work. Finally, Section 5.6 concludes and depicts on-going and future research work.

Part of the discussion and design proposal in this chapter was published in [Courtès *et al.* 2007b].

5.1. Introduction

In Chapter 2, we outlined the design of an *open*, cooperative backup service for mobile devices. In the general case, anyone is free to participate in the service and, therefore, the majority of participants have no prior trust relationship. There are also specific scenarios where owners of a few cooperating devices are personal acquaintances with full trust relationships as far as the backup service is concerned (e.g., colleagues, friends, etc.). An open cooperative service must be able both to account for the lack of prior trust relationships among participants and to take advantage of prior trust relationships among device owners when they exist.

Threats in the generic case where participants are mutually distrustful were outlined in Section 2.2:

- threats to confidentiality and privacy;
- threats to data integrity and authenticity;
- threats to availability.

Some of these threats have been already addressed by the storage layer presented in Chapter 4, while others can be addressed through data replication as was discussed in Chapter 3. However, *denial of service* (DoS) attacks committed by malicious participants have not been addressed so far.

denial of service

DoS attacks can lead to data or service unavailability. As such, they are effectively *threats to cooperation.* We listed in Section 2.2 the most obvious possible DoS attacks against our cooperative backup service or its users:

- *data retention*, where a contributor either refuses to send data items back to their owner when requested or simply claims to store them without actually doing so.
- *flooding*, where a participating device purposefully seeks to exhaust the service's shared resources.
- *selfishness* or *free-riding*, where users unfairly benefit from the cooperative service without contributing in return, thereby depriving the community of resources.

These attacks are well-known in resource sharing systems and cooperative services. They have been largely studied in the framework of Internet cooperative services such as peer-to-peer file sharing [Grothoff 2003, Cornelli *et al.* 2002, Lai *et al.* 2003] and cooperative backup [Lillibridge *et al.* 2003, Cox & Noble 2003, Aiyer *et al.* 2005]. Research on mobile *ad hoc* packet routing protocol also gave rise to similar research [Buttyán & Hubaux 2003, Buchegger & Boudec 2003, Michiardi & Molva 2002]. Selfishness attacks, in particular, have been the topic of abundant research since they are likely to occur if participants lacking incentives to contribute behave "rationally" [Aiyer *et al.* 2005]. In politics, as well as in the field of peer-to-peer computing, this phenomenon is often referred to as the *tragedy of the commons*, named after Garrett Hardin's famous essay on the problem of human population growth [Hardin 1968].

In this chapter, we identify fundamental requirements that must be fulfilled to allow such threats to cooperation to be tackled. We advocate that *accountability* is a key requirement. We propose *self-organized* security mechanisms that may be used to support behavior accountability and a wide range of cooperation policies. We show how cooperation policies can take advantage of these mechanisms to address some of our security concerns and discuss their integration with the security mechanisms already provided by the storage layer. Our approach differs from earlier work in that it focuses on *policy-neutral* security primitives that do not restrict the user's choice of a policy, rather than focusing on a specific policy.

policy-neutral

5.2. Architectural Overview of the Storage Layer

The storage layer presented in Chapter 4 addresses the efficient storage and indexing of data owners' critical data, following an append-only storage model. A simple RPC-based storage protocol that may be used among participating devices was also outlined in Section 4.3.3. Here, we focus on security aspects related to the storage mechanisms proposed in Chapter 4.

5.2.1. Core Storage Protocol

The storage framework discussed earlier can be summarized as follows:

1. The data owner (rather: the cooperative backup software on the owner-side) chops the data items to be backed up into small blocks and assigns them a *block name* (e.g., a cryptographic hash of the block contents). Important requirements are that (i) the naming scheme must be meaningless to contributors and (ii) blocks must be encrypted. In other words, contributors cannot make any assumptions on the block naming scheme used by data owners.
2. The data owner produces meta-data blocks describing, among other things, how data blocks are to be re-assembled to produce the original data. Those meta-data blocks are themselves named in a similar way. Meta-data blocks may also be encrypted to protect data confidentiality.

The end result of this backup process is an opaque identifier that names an (encrypted) root meta-data block. We refer to this identifier as the *root block name*. In addition, data blocks are transferred as follows: root block name

- When a contributor is encountered, the data owner sends it some of its data and meta-data blocks using remote procedure calls (RPCs), as discussed in Section 4.3.3. This is realized through the invocation `put(name, content)` which sends data `content` to the contributor and asks it to bind it to `name`. Since owners can choose any block naming scheme, contributors must arrange to provide *per-owner block name spaces* in order to avoid collisions among blocks belonging to different owners. Obviously, in order to increase data availability, data owners may choose to replicate each block, as seen in Chapter 3.
- When a contributor gains Internet access (rather, when it gets sufficiently cheap or high-bandwidth Internet access), it transfers data blocks stored on behalf of other devices to an Internet-based storage server that data owners can eventually access to restore their data.

Data and meta-data blocks must be *encrypted*. It must be possible for a data owner to *authenticate* them upon recovery, to make sure data being restored is genuine. The former could be achieved through public-key cryptography, using the data owner's public key to encrypt every block. However, this approach may be CPU-intensive. An alternative would be to use *convergent encryption* [Cox *et al.* 2002]. With this technique, data is encrypted with a symmetrical encryption algorithm, using the clear text's hash as the encryption key, as illustrated on Figure 31[1]. Since it relies on symmetric encryption, this method is much less CPU-intensive than public-key encryption [Menascé 2003][2]

convergent encryption

Authenticity is achieved by signing part of the meta-data. For instance, if meta-data blocks are the intermediate nodes of a Merkle hash tree whose leaves are data blocks [Merkle 1980], then only the root block needs to be signed, which reduces reliance on CPU-intensive cryptography; verifying the root block's signature actually allows the authenticity of the whole tree to be checked (see Section 4.2.4). In Chapter 6, we elaborate on the actual scheme that we implemented.

The root block name is critical since it allows all the user's data to be recovered, so it also needs to be backed up. However, as new versions of the data items (e.g., a single file or a whole file system hierarchy) are backed up, new data and meta-data blocks are created, each having a new name, and thus a new root block name is produced (this issue is not uncommon in the context of peer-to-peer file sharing and archival systems [Bennett *et al.* 2002, Quinlan & Dorward 2002]). Consequently, data owners should store their latest root block name on contributors *under a fixed block name* to allow restoration to be bootstrapped conveniently. Since it is a critical piece of information, data owners may choose to encrypt it.

Restoration of backed up data typically occurs when the data owner device has failed or been lost. In this case, data owners first retrieve the root meta-data block (from the Internet-based store), decrypt it and decode it (which can only be done by its legitimate data owner), and then recursively fetch the blocks it refers to. Fetching blocks upon restoration is achieved through a `get(name)` RPC that returns the contents of the block designated by `name`.

It is worth noting that among the mechanisms presented so far, only the actual storage protocol (i.e., the `put` RPCs) is imposed. This leaves users free to choose any *security policy* for their data: they may choose any data availability, confidentiality and integrity mechanism while still conforming to the storage protocol.

[1] Combining it with content-based indexing yields so-called *content-hash keys* or CHKs [Bennett *et al.* 2002, Clarke *et al.* 2001].

[2] Menascé notes that « symmetric key encryption is orders of magnitude fast than public key encryption » [Menascé 2003].

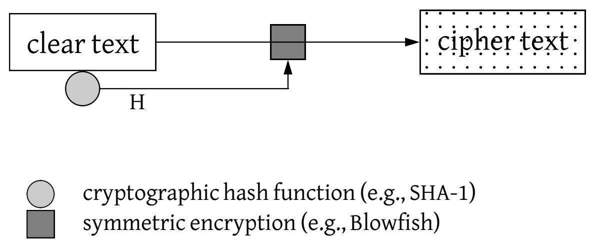

clear text
cipher text
H
cryptographic hash function (e.g., SHA-1)
symmetric encryption (e.g., Blowfish)

Figure 31. Convergent encryption with symmetric ciphers.

5.2.2. Interacting with the Internet-based Store

So far, we focused on the owner-side data and meta-data encoding/decoding, as well as on the storage protocol used by participating devices. A key assumption of our cooperative backup service is that participants can rely on a reliable Internet-based store as an intermediary between contributors and data owners for recovery purposes (Section 2.3). We do not focus on the implementation of such a store. Consequently, we ignored *how* it is implemented and remained open to a variety of possible solutions, ranging from per-owner stores to a single large shared store (e.g., an FTP-like server or peer-to-peer store known *a priori* to all participants). Nevertheless, how participants interact with such a store can influence the storage layer and its security mechanisms and is worth considering.

Suppose a shared store is used to receive data blocks from all contributors. Since it receives blocks belonging to different data owners that may use conflicting block naming schemes, such a shared store also needs to maintain per-owner block name spaces. Thus, when sending data blocks to the Internet store, contributors would need to specify who their owner is, e.g., by providing the data owner's "identifier". Similarly, upon recovery, a data owner would need to specify its name space by providing its identifier.

Of paramount importance is the inability for arbitrary users to tamper with a data owner's name space on the Internet-based store. For instance, it must be impossible for a malicious user to overwrite a data owner's block associated with a specific name on the Internet repository without this being detected. When data and meta-data encoding is owner-specific and unknown to the Internet-based store, said store cannot check the authenticity of incoming data blocks. Several approaches can be imagined to allow the Internet-based store to authenticate incoming data blocks:

- The Internet-based store could keep a list of all incoming data blocks associated with a given name, should different blocks be `put` under the same name (collisions). Upon recovery, the data owner can then decode and detect invalid data blocks in cases of collisions. In this case, the Internet store can remain oblivious to the owners' meta-data encoding schemes. However, this approach would make the Internet-based store vulnerable to flooding attacks.

- Requiring data owners to sign each (meta-)data block with their own private key does solve the problem, since it allows any third party to authenticate each block. However, this solution is not acceptable since it would involve a lot of costly cryptographic operations.
- The meta-data scheme could be fixed and used by all data owners or somehow communicated to the Internet-based store. This would allow the Internet store to authenticate data on behalf of data owners, at the cost of reducing the flexibility available to data owners in terms of meta-data structures.

This last solution appears to be the most acceptable for setups with an Internet-based store shared among data owners. Furthermore, it can be implemented in such a way that the Internet-based store is able to authenticate data blocks without being able to access its contents.

5.3. Leveraging Cooperation

In this section, we present our approach to the design of mechanisms that address the threats to cooperation summarized in Section 5.1. Core mechanisms are proposed to support accountability while being neutral with respect to cooperation policies. We then discuss issues that arise from the self-organized nature of our approach as well as cooperation policies.

5.3.1. Design Approach

There are essentially two ways to provide security measures against the DoS threats listed earlier in MANETs and loosely connected peer-to-peer backup systems: *via* a *single-authority domain*, where a single authority provides certificates or other security material to participants and/or imposes a particular policy or mechanism, or through *self-organization*, where no single authority is relied on, at any point in time [Capkun *et al.* 2003].

self-organization

In our opinion, reliance on a common authority responsible for applying external *sanctions* to misbehaving participants as in BAR-B [Aiyer *et al.* 2005] falls into the first category. For example, BAR-B contributors *must* provide a proof that they do not have sufficient space when rejecting a storage request; similarly, upon auditing, participants *must* show the list of all blocks stored on their behalf elsewhere and all blocks they store on behalf of other nodes. Failing to do so constitutes a “proof of misbehavior” that may lead to sanctions. This raises fundamental security issues: why would one disclose all this information to some supposedly trusted entity? Does it still qualify as cooperation among *multiple* administrative domains when a single set of rules is enforced through external sanctions? While this approach achieves strong service provision guarantees, it does so at the cost of being authoritarian and seems unsuitable for the kind of open cooperation network we envision.

Likewise, the use of so-called "tamper-resistant security modules" as in [Buttyán & Hubaux 2000] can be considered as a single-authority domain approach: security modules act as a local representative of an "authority" and *enforce* part of the protocol (in [Buttyán & Hubaux 2000], the *nuglet* mechanism) in order to provide protection against malicious users. This leaves the user with no choice but to abide by the rules set forth by the security module and the party that issued it.

We are of the opinion that reliance on a central authority can in itself be considered as a security threat, to some extent: that authority is in effect a *single point of trust* and its compromise would bring the whole service down. Furthermore, depending on their security policy, users may not be willing to fully trust such an authority just because they have been told it's a "trusted" authority. They may also want to have full control over the actions that can be taken by *their* device, rather than handing over some authority over the device to some possibly unknown third party. Also, since we are designing an *open* cooperative service where anyone can participate, self-organization is likely to make the service more readily accessible to everyone; conversely, requiring every user to register with some central authority would be an undesirable burden likely to limit user adoption. Therefore, we do not consider solutions based on a single-authority domain but prefer to focus on totally self-organized solutions. Indeed, such solutions are a good match for mobile *ad hoc* networks which *are* self-organized.

As a consequence, we cannot assume that any single cooperation policy is going to be used by *all* devices: each device can, and will, implement its own policy. We believe that the ability to choose a security and cooperation policy is particularly important when using our cooperative backup service for two reasons. First, the goal of this service is to improve the availability of users' critical data. As such, users are likely to be willing to pay attention to the contributors they deal with, and hence, they may be concerned with their cooperation policy. Second, mobile devices being resource-constrained, users are likely to require tight control over their resource usage, and may want to implement a cooperation policy that makes the best use of their resources. This is quite different from, for instance, Internet-based file sharing services where participating devices are typically desktop machines and where, as a result, it is safe to assume that most users will be satisfied with the same default cooperation policy.

accountability

A key observation is that most cooperation policies rely on *accountability* [Dingledine *et al.* 2001]. To enable cooperation among untrusted principals, it must be possible to hold participants accountable for their actions, such as their resource consumption, their contributions, as well as their misbehaviors. Without this, a decentralized system is obviously vulnerable to all sorts of abuses, which eventually leads to denial of service. We view accountability as a core mechanism that is a prerequisite to the implementation of cooperation policies. Therefore, in this chapter we focus on core mechanisms allowing for accountability rather than on actual cooperation policies.

5.3.2. Providing Verifiable and Self-Managed Device Designation

Devices must be able to *name* each other (i) to achieve accountability and (ii) to allow contributors to implement per-owner block name spaces, as discussed in Section 5.2.

To these ends, device names must satisfy the following requirements. First, since we want to build a self-organized service, where no central authority has to be consulted, it must be possible for every device to create its own name or designator. Second, for the naming scheme to be reliable, device names must be *unique* and *context-free* (i.e., their interpretation should be the same in any context). Third, since device names serve as the basis of critical operations, it must be possible to *authenticate* a name-device binding (i.e., assess the legitimacy or "ownership" of a name). Authentication is needed to preclude unauthorized use of a name, as in *spoofing* attacks. Unauthorized uses of device names would effectively hinder the implementation of per-owner block name spaces and accounting mechanisms.

These requirements rule out a number of widespread designation mechanisms. IP addresses, for instance, would obviously be unsuitable to name devices since they have none of these properties (they are not context-free, especially IPv4 link-local addresses, not unique, except for IPv6 addresses, and cannot be authenticated). The designers of Mobile IPv6 (MIPv6) had similar requirements and had made the same observations. This led them to devise "statistically unique and cryptographically verifiable" (SUCV) addresses [Montenegro & Castelluccia 2002].

The building block for the naming scheme we are interested in (and that of MIPv6 SUCV addresses) is asymmetric cryptography. Public keys have all the desired properties as designators: they are (statistically) unique and context-free, and they provide secure naming (i.e., the name-device binding can be authenticated, thereby precluding spoofing). In practice, public keys can be too large to be used directly as designators, which is why several protocols use cryptographic hashes or fingerprints of the public keys as designators [Callas *et al.* 1998, Montenegro & Castelluccia 2002]. In order to achieve accountability, both contributors and data owners may wish to *identify* the device they are talking to, that is, to authenticate the binding between the alleged name of the peer device and the device itself. In other words, *mutual authentication* is required.

mutual authentication

It is worth noting that the entities we want to name are instances of the cooperative backup software running on participating devices and *not* people owning the devices, nor even physical devices. Thus, the principals involved in the cooperative backup service are *logical entities* that exist and interact solely through electronic interactions among them. Therefore, authenticating the binding between one of these entities and its name (public key) boils down to verifying that this entity holds the private key corresponding to its name [Ellison 1996]. Doing so is simple and does not require the use of any certification authority whatsoever.

As far as the data restoration bootstrap is concerned, a practical consequence of using public key pairs to identify devices is that a user's key pair is all that is needed to bootstrap restoration, assuming its public key is also used to encrypt the root block name. That means that users must store their key pairs reliably, outside of the cooperative backup service, by copying them on a storage device under their control (USB stick, computer, or even a simple piece of paper stored in a safe place). Obviously, the device where the user's key pair is stored must not be carried along with the mobile device itself, since it could easily be lost, stolen, or damaged along with the mobile device, making it impossible to recover the data. Elliptic curve cryptography (ECC) would be handy for that purpose: it yields keys much smaller than, e.g., "security-equivalent" RSA keys; thus an ECC key pair can be as simple as a pass phrase that may be readily memorized by the user.

5.3.3. Ensuring Communications Integrity

Once a participating device has authenticated the binding between a peer device and a name, a malicious device may try to send messages and pretend they originate from another device, thereby using resources on behalf of another device. To address this issue, the *integrity* and *authenticity* of messages (i.e., RPC invocations) that devices exchange must be guaranteed by the communication layer; in other words, it must be practically impossible for an attacker to modify the message payload or its meta-data (e.g., information about the source and recipient) in an undetectable way. In particular, once devices have mutually authenticated, using their key pairs, the communication protocol must guarantee that messages received at either end of the communication channel still come from the previously authenticated device. Many well-known cryptographic protocols address this issue, with different security properties.

We believe that *non-repudiation*, i.e., the ability for a third-party to verify the integrity and authenticity of a message, is not required in our decentralized, self-managed, cooperative backup system. Non-repudiation could be used, for instance, to make sure that a device cannot deny that it sent a series of storage requests to a certain contributor. That contributor could then *prove* to a third party that it did receive those requests. However, such proofs would likely not be sufficient to be used, for instance, as part of the "history records" maintained by a reputation system (described below): they would concern only individual requests and would consequently fail to provide a sufficiently high-level view of a device's past cooperation. For instance, to prove that a data owner requested 1 GiB of storage, a contributor would need to provide a third party with 1 GiB worth of *put* requests along with the corresponding signatures. Doing so would provide more information than is necessary and would be very bandwidth-consuming, making it impractical. Thus, non-repudiation of individual messages is inappropriate in our context.

Therefore, we plan to use regular message authentication codes (such as HMACs) to provide support for message authenticity checks. HMACs can only be verified by the receiver, and therefore do not provide non-repudiation.

5.3.4. Thwarting Sybil Attacks

Sybil attack

Since key pairs are to be generated in a self-organized way, our system is subject to the *Sybil attack* [Douceur 2002, Marti & Garcia-Molina 2003]: devices can change names (i.e., public keys) any time they want, which allows them to escape accountability for their past actions, including misbehavior. Successful Sybil attacks can defeat any resource accounting mechanism and resource usage policy. For instance, a data owner can completely circumvent a per-device quota implemented by a contributor by just switching to a new key pair.

The verifiable designation mechanism proposed above cannot by itself prevent Sybil attacks. Instead it is up to cooperation policies to make Sybil attacks less attractive by providing incentives for users to keep using the same name (i.e., the same key pair). In a system where names are managed in a self-organized way, no cooperation policy can *prevent* Sybil attacks: They can only make them less effective. However, evidence shows that well-designed policies can make them pretty much worthless [Marti & Garcia-Molina 2003, Michiardi & Molva 2002, Buchegger & Boudec 2003].

Naturally, most reasonable cooperation policies have a common denominator: they tend to be reluctant to provide resources to strangers while being more helpful to devices that have already cooperated. However, in order to bootstrap cooperation, many policies may grant at least a small amount of resources to strangers [Grothoff 2003]. This means that there is usually (i) a medium- to long-term advantage in keeping the same name and (ii) a short-term advantage in cooperating under a new name. Section 5.3.5 will show how actual cooperation policies can achieve this.

Fortunately, the impact of Sybil attacks is largely a matter of scale. With Internet-based peer-to-peer cooperative services, any peer can reach thousands of peers in a glimpse. Thus, even if it can only benefit from a small amount of resources from each peer, it may be able to quickly gain a large amount of resources. Conversely, in a cooperative service relying on physical encounters among mobile devices, it may take a long time and a great deal of traveling around before one is able to gain access to a useful amount of resources, which effectively makes selfishness less viable economically. Likewise, the impact of a flooding attack is necessarily limited to physical regions and/or groups of devices. Furthermore, if there are few resources to be gained through Sybil attacks, then there is an incentive to keep using the same name, which in turn provides an incentive to honor promises to store data. Of course, it is up to cooperation policies to make such attacks even less attractive by providing additional incentives for cooperation.

5.3.5. Allowing for a Wide Range of Cooperation Policies

User cooperation policies define the set of rules that determine how their device will cooperate. They are usually concerned with the stimulation of cooperation and the establishment of trust with other devices. To that end, cooperation policies can build on the accountability provided by the mechanisms presented above. We can imagine two major classes of cooperation policies: those based on the underlying *social network*, and those based on past *behavioral observations*, either through private or shared "history records" [Grothoff 2003, Lai *et al.* 2003, Michiardi & Molva 2002, Buchegger & Boudec 2003]. It is our goal to allow users to choose among these cooperation policies.

social network

behavioral observations

Cooperation policies based on the relationships already existing in the underlying social network can be as simple as "white lists", where the user only grants resources to devices belonging to personal acquaintances. There can also be more sophisticated policies: a user could also accept storage requests from "friends of friends", and it could accept to dedicate a small amount of resources to strangers as well. It can be argued that such policies do not scale since (i) the number of personal acquaintances of an individual is limited, and (ii) when travelling a lot, these acquaintances may be out of reach. On the other hand, social studies have provided evidence of a "small-world phenomenon" in human relationships [Milgram 1967, Capkun *et al.* 2002] and algorithms have been proposed to discover chains of acquaintances among arbitrary users [Capkun *et al.* 2003]. These studies can make cooperation policies based on a social network more relevant. Such policies, were they to insist on being able to verify bindings of keys to real-world identities, would trade privacy for improved resilience to Sybil attacks. However, similar policies may be used with pseudonyms instead of real-world identities.

Cooperation policies based on observations of past device behavior provide an interesting alternative: devices maintain "history records" of each other and make cooperation decisions using them as an input. History records can either be local to a device or they can be shared among devices—the latter is usually referred to as a *reputation* system [Lai *et al.* 2003, Buchegger & Boudec 2003, Michiardi & Molva 2002]. In a reputation system, devices exchange history records or opinions and may use them as an additional hint to their cooperation decisions. Simulations have shown that shared history records are usually more efficient than private history records, especially in large networks or in the presence of a high device turnover [Lai *et al.* 2003, Buchegger & Boudec 2003][3]. Reputation mechanisms make Sybil attacks unattractive since few resources can be gained by a stranger.

reputation

However, many works that evaluate the outcome of such reputation mechanisms assume that all participating nodes use the *same* cooperation policy [Michiardi & Molva 2002, Buchegger & Boudec 2003] (e.g., the same node rating algorithm, the same decision-making algorithm, etc.). There is no reason for this to be true. The result of using a reputation mechanism in a world where different policies are in use is, to our knowledge, an open issue. For example, when all participants have the same cooperation policy, an adversary could exploit its weaknesses in a systematic way, as illustrated by *BitThief* [Locher *et al.* 2006]. Conversely, it would be harder for an adversary to devise a viable strategy in the presence of multiple policies, and without additional information about the set of policies. Analytical evaluations based on game theory, e.g., as used in [Oualha *et al.* 2007a], appear to

[3] See Section 5.5 for additional information.

be better suited to situations with diverse cooperation policies as they focus on individual interactions between peers rather than on the algorithms running at each peer.

While leaving users the freedom to choose their cooperation policy increases the system's flexibility, it may also reduce its scalability if participants are unable to help each other make informed cooperation decisions. One solution would be to allow participants to exchange trust information about each other. However, to our knowledge, this is largely an open issue: just like humans, participants may have different trust metrics and different ways to assess and reason about trustworthiness, which makes it difficult to think of a formal way to "convert" trust information at peer boundaries.

From a privacy viewpoint, maintaining history records may be a concern when identities are bound to real-world entities, since it would allow one to know where a given person was at a given point in time. However, for users' privacy to be seriously threatened, attackers would need to *physically* track them, which the cooperative backup service could hardly be held accountable for. This is a lesser concern when identities are not bound to real-world entities.

5.4. Implementation Considerations

This section discusses implementation concerns and in particular the choice of actual protocols to achieve the goals outlined earlier.

5.4.1. Protocol Choice

While Mobile IPv6 [Montenegro & Castelluccia 2002] provides some of the features we need, we considered it impractical since its mechanisms are implemented at the network layer, and implementations are not widely available at this time.

Our implementation of the block store (essentially the `put` and `get` requests mentioned earlier) is based on Sun/ONC RPC [Srinivasan 1995]. ONC RPC defines the so-called "DES authentication mechanism", designed for authentication over a wide-area network; however, the mechanism does not address all our concerns (for example, its naming scheme for peers does not fulfill all the requirements of Section 5.3.2, and in particular does not allow name-device bindings to be reliably authenticated). The authentication mechanisms for ONC RPC defined in RFC 2695 [Chiu 1999] have similar shortcomings with respect to our goals. The RPCSec bindings for the Generic Security Services Application Programming Interface (GSS-API) [Eisler *et al.* 1997] were not considered appropriate either (one reason is that most available GSS-API implementations only support Kerberos-based mechanisms, which assumes the availability of such an infrastructure).

Consequently, we decided to use the well-known Transport Layer Security (TLS), a protocol currently widely deployed on the Internet [Dierks *et al.* 2006]. Although it was not designed with mobile computing and constrained devices in mind, we believe its flexibility makes it a suitable choice. In particular, TLS offers a wide range of *cipher suites*, which allows us to choose cipher suites that meet our resource saving constraints, such as cipher suites with no payload data encryption, as discussed in Section 2.2.1. TLS provides message authentication guarantees using HMACs, where, again, the HMAC algorithm to be used is negotiated between peers. TLS provides payload compression but this may be disabled (also subject to negotiation between peers). Again, disabling it allows us to save energy, especially since the data that is to be exchanged among peers is already compressed (see Chapter 4).

As far as mutual authentication is concerned, TLS provides it through *certificate-based authentication mechanisms.* While the main document [Dierks *et al.* 2006] refers primarily to X.509 certificates, an extension adds support for authentication using *OpenPGP certificates* [Mavrogiannopoulos 2007]. This extension is very relevant in our context for a number of reasons. First, OpenPGP certificates can be readily generated using widely available tools (e.g., GnuPG) and they are already familiar to many computer users. Second, OpenPGP certificates are already used in the context of secure electronic communications among individuals. Therefore, the use of OpenPGP certificates also allows users to easily implement cooperation policies based on the underlying social network, as outlined in Section 5.3.5.

OpenPGP certificates

OpenPGP certificates contain a lot more than just a public key. In particular, since they are primarily used to certify a binding between a public key and a real-world person name, they contain information such as the real-world name and email address of the person the public key (allegedly) belongs to (the "user ID packets"), and a list of third-party signatures (certifications) indicating the level of trust put by other people in this name-key binding [Callas *et al.* 1998]. This information is only useful when implementing cooperation policies based on the social network.

5.4.2. Preliminary Prototype

We have been working on a prototype implementation of our cooperative backup protocol that uses ONC RPC on top of TLS, namely by extending our *libchop* remote block store presented in Section 4.3.3. Since ONC RPC implementations do not natively support the use of TLS as the underlying protocol, we did our own implementation. This proved to be easy to do, using raw TCP RPC client/server code from the GNU C Library as a starting point. We use GnuTLS [Josefsson & Mavrogiannopoulos 2006] as the underlying TLS implementation since it is the only major implementation supporting the OpenPGP extension [Mavrogiannopoulos 2007] as of this writing. GnuTLS is very flexible and has allowed us to actually make various specific trade-offs, such as disabling compression, choosing an encryption-less cipher suite, etc.

Initial measurements show that TLS induces little communication overhead. Handshake itself demands 2 KiB per connection in both directions (when using certificates with no signature packets), most of which stems from the OpenPGP certificate exchange. The TLS record layer incurs little overhead (e.g., less than 30 octets per message with SHA-

1-based HMACs), provided messages are at most 16 KiB long—otherwise messages are fragmented, which incurs additional overhead [Dierks *et al.* 2006]. Although further measurements are needed, these results seem reasonable in our context.

5.5. Related Work

This section gives an overview of related work that addresses denial-of-service attacks against cooperative services similar to ours. We then discuss common cooperative incentives described in the literature, and finally related work on designation in distributed and self-organized systems.

5.5.1. Denial of Service Attacks on Cooperative Services

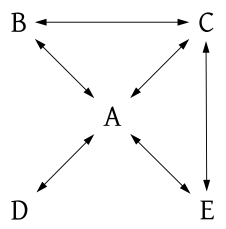
A lot of work has gone into thwarting availability threats due to DoS attacks similar to those described in Section 5.1. Most of this work was done in the area of peer-to-peer storage and cooperative backup [Lillibridge *et al.* 2003, Cox & Noble 2003, Grothoff 2003]. Dingledine *et al.* wrote an excellent classification and summary of existing approaches to tackle DoS attacks in peer-to-peer systems deployed on the Internet [Dingledine *et al.* 2001]. They acknowledged accountability as an enabler for cooperation among distrustful principals. Understandably, the operating system and programming language communities reached the same conclusion as computers went multi-user and as executing untrusted code became commonplace.

While our cooperative backup scheme with intermittent connectivity to the infrastructure is similar to delay-tolerant networks [Zhang 2006], the security of such networks is still largely an open issue [Farrell & Cahill 2006, Harras *et al.* 2007]. This is partly due to the fact that most applications of DTNs, such as space mission networks, are not expected to be open for anyone to participate, which reduces the incentive to address these issues.

Fall *et al.* did propose security mechanisms permitting DTN routers to detect and eliminate disallowed traffic, and thereby avoid DoS attacks such as flooding against the DTN [Fall 2003]. However, the proposed solution relies on centralized identity management and authorization: all participants are issued a key pair by an authority, along with a "postage stamp" signed by that authority indicating the allowed "class of service" for that user. Such an approach only addresses specific DoS attacks. Forms of non-cooperation such as refusal to forward a message are not tackled. We also believe that such an approach does not scale and suffers from shortcomings inherent to single-authority domain approaches, as discussed in Section 5.3.

5.5.2. Cooperation Incentives

In general, "trust begets cooperation". In the case of our cooperative backup service, data owners need to trust contributors to provide them the service, while contributors need to



Node	A	B	C	D	E
A	∞	100	3	0	42
B	0	∞	0	—	—
C	5	200	∞	—	2
D	1	—	7	∞	—
E	140	—	3	—	∞

Figure 32. Trust levels based on private history records in a small network [Grothoff 2003].

trust data owners not to abuse the service (e.g., by flooding it or by being selfish). While both issues have to do with trust establishment between owners and contributors, the literature tends to refer to both aspects using different names, such as *cooperation incentives* and *trust establishment.*

cooperation incentives

To evaluate the cooperativeness of a peer, one needs to be able to observe both its service usage and its service provision. When the cooperative service is packet forwarding or routing in MANETs, device cooperation can be evaluated almost instantaneously [Michiardi & Molva 2002, Buchegger & Boudec 2003, Grothoff 2003]. However, in cooperative backup services, service usage and service provision call for different evaluation techniques. First, service usage can be balanced using simple strategies such as symmetric trades [Lillibridge *et al.* 2003] (i.e., pairwise "tit-for-tat" exchanges), or "storage claims" that may be exchanged among peers [Cox & Noble 2003]. Both approaches assume high connectivity among peers and are therefore unsuitable for MANETs. Second, *periodic auditing* has been proposed to establish trust in contributor service provision [Lillibridge *et al.* 2003, Cox *et al.* 2002, Cox & Noble 2003, Aiyer *et al.* 2005], but this usually requires peers to be reachable so that they can be challenged, which is unsuitable to the MANET context.

periodic auditing

A novel auditing protocol has been proposed to overcome this limitation, allowing the auditing responsibility to be *delegated* [Oualha *et al.* 2007b]. Data owners can grant a set of chosen *verifiers* the ability to audit a specified contributor, and data owners do not need to disclosed the data to be audited to verifiers. However, it is unclear at this stage how verification results could be used as an input to cooperation policies. In particular, it is unclear how verification results could be authenticated by the data owner itself or by third-parties such as participants in the vicinity of a verifier and its assigned contributor.

Once service provision and usage can be evaluated, self-organized solutions usually make use of history records of peer behavior as an aid to cooperation decisions, as mentioned in Section 5.3.5. Figure 32 illustrates "trust levels" as computed by GNUnet nodes based on their own observations [Grothoff 2003]: the left-hand side of the figure shows connections between nodes, while each cell of the table on the right-hand side shows the level of trust of one node in another. From the figures, we see that trust levels based on private observations are neither transitive, nor symmetrical, nor homogeneous (e.g., C's trust in B is higher than A's trust in B).

Lai *et al.* showed through simulation that private history records (i.e., based solely on local information) do not scale well to large populations, unlike shared history records (i.e., reputation); however, they also note that reputation systems need countermeasures to address the risk of *collusion* among peers [Lai *et al.* 2003]. Similarly, other simulations showed that use of second-hand or even third-hand opinions allow misbehaving nodes to be detected faster, even in the presence of liars, and especially with large populations [Buchegger & Boudec 2003]; the authors also propose a way to merge partially trusted third-party opinions with one's own opinion. In MANETs, reputation mechanisms have been proposed primarily in the context of packet forwarding for multi-hop routing protocols and route discovery [Michiardi & Molva 2002, Buchegger & Boudec 2003].

5.5.3. Designation

Designation issues in a decentralized environment have been studied notably in the context of distributed programming and capability systems [Miller 2006] as well as in the context of public key infrastructures (PKIs) [Ellison *et al.* 1999, Ellison 1996]. The provision of guarantees for "address ownership" (i.e., having address-device bindings that can be authenticated) has also been a concern in the design of Mobile IPv6 (MIPv6) [Montenegro & Castelluccia 2002]. This led the authors to opt for "statistically unique and cryptographically verifiable (SUCV) identifiers". This is similar to one of the mechanisms we propose in this chapter, except that we operate at the application level rather than at the network layer, which provides us with more flexibility.

Douceur *et al.* described the Sybil attack as a problem that is inherent to distributed systems using self-managed designators [Douceur 2002]. In [Marti & Garcia-Molina 2003], the authors showed that a reputation system can efficiently leverage cooperation even when self-managed designators are used.

5.6. Summary

The contributions of this chapter can be summarized as follows:

- We identified security threats on a self-organized cooperative backup service for mobile devices and listed subsequent security requirements.
- We have shown that a reduced set of well-known cryptographic primitives can be used to meet these requirements in a self-organized way. Our approach differs from earlier work in that it focuses on *policy-neutral* security mechanisms, rather than on a specific cooperation policy.
- We proposed the use of public keys as self-managed, verifiable and unique designators for participating devices and discussed their use as a policy-neutral building block for a variety of cooperation policies, including a reputation system.

- Systems using self-managed designators are subject to the Sybil attack. We discussed the impact of this attack in our context and showed how cooperation policies can be implemented that reduce the harm that can be done.
- We discussed implementation issues and outlined the foundations of an implementation that uses TLS with OpenPGP certificate-based authentication. This augments the storage layer implementation presented in Chapter 4 with support for distributed storage among distrustful principals.

The next chapter describes our implementation of the cooperative backup service and shows how the elements brought in this chapter fit in.

Chapter 6. Implementation

So far, several key aspects of the cooperative backup service have been discussed. This chapter presents our implementation of the service, which draws from the results and proposals made in earlier chapters. Our implementation seeks to be a close match to the design goals stated in Chapter 2. It tries to closely follow the storage evaluation and proposals given in Chapter 3 and 4 as well as the security approach from Chapter 5.

While previous chapters provided important results and proposals for all of these aspects of the design, pieces are still missing to build a comprehensive cooperative backup service. This chapter aims to fill this gap. We first provide a brief overview of our implementation. Second, high-level storage facilities that complement those discussed earlier are presented. Third, the design and implementation of the various *activities* of our cooperative backup software are discussed: input data indexing, opportunistic replication, storage provision and data retrieval. Finally, a preliminary evaluation of the software is conducted.

6.1. Overview

Our cooperative backup software is written in less than 5,000 lines of Scheme code [Kelsey *et al.* 1998] for GNU Guile [Jaffer *et al.* 1996]. It makes use of *libchop*, the storage layer written in C and discussed in Chapter 4. The implementation consists of a *backup daemon* and a set of *clients* allowing the end-user to interact with it. The daemon implements most of the activities of cooperative backup, as they are all intimately related. These will be discussed in Section 6.3.

backup daemon

The choice of the Scheme programming language [Kelsey *et al.* 1998] was motivated by several aspects, beside the author's preference. First, Scheme's semantic clarity makes it easier to reason on the program's behavior, specifically when it is written in a purely functional style. Referential transparency, for instance, as well as the use of functional forms that combine functions to realize higher-level ones, greatly simplify the reasoning on algorithms such as those presented in Section 6.3 [Backus 1978]. Second, Scheme's support for higher-order functions, i.e., functions that take functions as arguments or that return functions, greatly benefits the flexibility of the system. For instance, most of the algorithms used by the backup service are parameterized by predicates or functions that implement some policy (e.g., replication policy, cooperation policy). Our goal is to provide users with the ability to parameterize the algorithms to fit their needs, which is easily achieved using higher-order functions.

The backup daemon needs to be provided with an OpenPGP key pair upon startup. This key pair will be used when interacting with other backup daemons, as discussed in Chapter 5. Further interaction with the user is done through an ONC RPC client that talks to the daemon over Unix-domain sockets. Our command-line client, called `ars-command`, allows users to schedule new files for backup and to tell the backup daemon about the IP address and port of other backup daemons. It also allows users to request *synchronous* backups, i.e., to force the backup daemon to take a snapshot of their collection of files. Users can annotate such snapshots, making it easier to look for them. This feature is similar to the `commit` operation found in revision control systems, for instance.

A set of command-line tools allow for data retrieval and access to the revision history of files. Again, these tools communicate with the backup daemon using ONC RPCs over Unix-domain sockets. They are discussed in further detail in Section 6.3.3.

journal

When experimenting with the software, we felt the need for the ability to visualize interactions among instances of the backup daemon. To that end, we designed a *journal* RPC interface that allows backup daemons to notify some remote monitor of cooperation events that occur. Such events include the discovery of a participating device, the initiation or acceptance of a connection with another device, the sending or reception of data blocks to or from another device. Frédérick Capovilla and Guillaume Vachon implemented in Java a graphical user interface, which we call *MERLIn*[1], that listens for such event notifications coming from an arbitrary number of backup daemons and translates them into a concise graphical representation. MERLIn can also record a sequence of events, along with timestamps, which makes it easier to understand the interactions among backup daemons.

6.2. High-Level Storage Facilities

The storage layer presented in Chapter 4 provides the basis for an atomic append-only block store. Specifically, it allows arbitrary input data streams to be chopped into smaller blocks where:

- each data block is assigned a unique *name* or *index* that is used to store it to (retrieve it from) a *keyed block store*;
- suitable *meta-data blocks* designating the blocks that must be fetched to restore the original data stream are created; they are stored like regular data blocks;
- finally, a single block name designating the "root" meta-data block of a data stream suffices to restore said stream.

In addition, our implementation of this storage layer, *libchop*, provides support for lossless compression, data integrity checking through cryptographic hashes, and encryption.

[1] MoSAIC Event Listener and Recorder Interface

Three components not present in this storage layer are needed by the cooperative backup software. First, the cooperative backup software operates on files rather than on anonymous input data streams. Thus, it needs to store information about *file meta-data* . Second, as subsequent revisions or "snapshots" of each file may be created and stored locally during the cooperative backup process, *revision meta-data* must be produced. Finally, a mechanism must be devised to allow full recovery to be bootstrapped. The mechanisms we have implemented are described below.

file meta-data

revision meta-data

6.2.1. Directories

The input of our cooperative backup service is essentially a list of *file names*. The storage pipeline of Chapter 4 is applied to the contents of these files[2], which yields one *index* per file. Each index can then be used to retrieve the contents of the corresponding file. From the user's viewpoint, being able to restore a file from its name, or being able to restore an entire directory hierarchy (including proper file names) is an obvious requirement. Thus, a mapping between file names and indices must be provided. Of course, this mapping must also be replicated, just like file contents, so that it can be restored upon recovery.

Our cooperative backup software handles this by storing and replicating the list of file-name/index pairs. We refer to such a list as a *directory*. Directories do not have a special status in our implementation: they are simply a textual representation of the file-name/index pairs and are stored using the same storage pipeline as other input data streams.

directory

For each snapshot of the user's data, a single directory is created. It lists all the files and indices of the user's files subject to backup, rather than just the files that were added or modified since the last snapshot. The rationale is that the directory representing the last snapshot contains all the information needed to retrieve the last version of *all* user files, which allows the index of any user file to be retrieved in constant time (i.e., independently of the number of revisions that have been stored). This approach is similar, for instance, to the "tree" structure used by the Git revision control system [Hamano 2006].

Therefore, if a new directory is stored every time a file is scheduled for backup or modified, two subsequent directories may have a lot of entries in common. Consequently, since directories are to be stored using the storage layer of Chapter 4, including mechanisms providing single-instance storage, storing a new directory that is very similar to previous ones requires little additional storage. We could imagine further optimizing sharing across similar directories by using one data block per directory entry. However, this would increase the amount of meta-data for each directory.

[2] The components of the storage pipeline (e.g., chopper, block indexer) can be specified by the user. By default, the backup daemon uses CHK block indexing, which provides block encryption without being too CPU-intensive (see Section 5.2).

6.2.2. Revision Chains

As stated earlier, users may produce new data before previously created data has been entirely replicated. Thus, the cooperative backup software needs to support some form of *versioning*. Namely, a new "snapshot" (or revision) of the user's files must be created when file contents have changed or the directory layout has been altered (e.g., new files have been created). As we have just seen, each snapshot is represented by a new directory structure that lists pairs of file names and corresponding indices. In addition to directories, another data structure allowing available revisions to be browsed appears to be essential.

versioning

To that end, we opted for a singly linked list whose head is the latest revision. Each revision contains the index of the previous revision, the index of the directory corresponding to this revision (noted `root`), along with meta-data describing the revision (e.g., date of the revision, optional user-specified log message). Figure 33 illustrates the chain of revisions (square boxes) along with the corresponding directories (rounded boxes); for file `/src/chbouib.c`, the figure also illustrates how these data structures are linked to the stream data (i.e., file contents) and meta-data as presented in Chapter 4. Arrows on the figure represent pointers from one data structure to another. Such pointers are actually implemented as *block indices* as shown in Chapter 4. Our data structure for revisions is similar to the "commit" data structure found in the Git revision control system [Hamano 2006]. It is biased toward optimized accesses to the latest revision (i.e., the head of the list), which is independent of the number of revisions.

6.2.3. Root Index

In Section 5.2, we already mentioned the "root block name" or "root index". In this storage framework, the root index is the index of the latest revision structure, represented by the rightmost arrow, labeled "(head)", on Figure 33. The root index is the only data structure that needs to be modified when a new revision is created: the rest of the storage strictly follows an append-only model. Our implementation stores the root index under a fixed name, which allows it to be retrieved without additional knowledge about the available snapshots. This permits recovery from a catastrophic failure.

We also mentioned in Section 5.2 that the root index is sensitive data (since it allows all the user files to be retrieved) that needs to be protected from unauthorized access. This is achieved by signing and encrypting it using the user's key pair. Thus, the user's key pair suffices to restore all the user's data[3]. Upon recovery, the root index is deciphered and authenticated.

[3] Conversely, a single file inserted into the user's block name space is accessible without knowing the user's key pair. However, the file's index and encoding parameters need to be known to permit retrieval. Since block indices such as 160-bit hashes are unguessable, this effectively provides "protection by sparsity" where *knowing* a file's root index suffices to access it [Tanenbaum *et al.* 1986]. File sharing scenarios could be imagined where a principal communicates the root index of one of their files to a contributor, thereby allowing the contributor to retrieve the file.

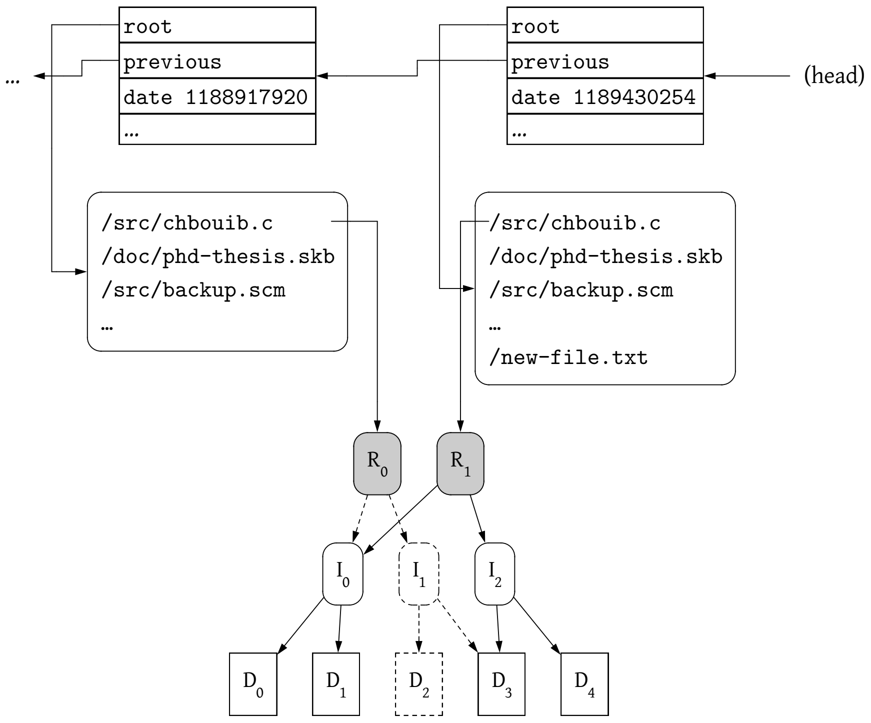


Figure 33. Revision chains, directories and file contents.

6.3. Cooperative Backup Activities

Our backup daemon implements actual data backup (i.e., input data "chopping" and indexing using *libchop*, as well as opportunistic replication), data retrieval, and storage provision (i.e., contributing storage to participating devices). The following sections provide an overview of the algorithms used for each of these activities.

6.3.1. Input Data Indexing and Block Queuing

As explained in Chapter 2, input data must be chopped into small blocks before it can be transferred to contributors. This is done using a storage pipeline similar to the ones described in Section 4.3 as far as input stream indexing is concerned, and using the data structures described in Section 6.2 as far as file system meta-data is concerned.

Rather than waiting for a contributor encounter, our cooperative backup daemon proceeds with input data chopping and indexing as soon as new files are scheduled for backup or existing files subject to backup are modified. Therefore, data and meta-data blocks that must be replicated are already available when a contributor is encountered, allowing the daemon to quickly react to contributor discoveries. This *pre-indexing* of input pre-indexing data is made possible by the fact that the encoding used by the storage pipeline does not depend on the contributor that will receive the data blocks. This approach is similar to that of Pastiche where regular file systems are layered on top of a "chunk store" that effectively maintains a list of data and meta-data blocks to be replicated as user files are modified [Cox *et al.* 2002].

The indexing process is tightly coupled with opportunistic replication. Blocks resulting from the indexing of new user data must be marked for eventual replication. Our implementation achieves this by associating meta-data with each block subject to replication. This meta-data lists contributors already holding a *replica* (more precisely, it lists the 20-byte OpenPGP fingerprint of each such contributor). This information is used as an input to the opportunistic replication algorithm, as we will see later. In addition, in our current implementation, the meta-data associated with a block also contains a "reference counter" indicating how many times the block was referenced; this gives an additional hint that could drive the replication algorithm (e.g., by creating additional replicas of highly-referenced blocks). Although we have not yet implemented it, the user-defined priority or desired level of replication of a given block could be derived from that of the corresponding input file and stored alongside the other block meta-data; this would allow users to provide hints directly usable by the replication algorithm.

replica

Technically, keeping track of these data blocks is achieved by using an *ad hoc* implementation of the `block_store` interface presented in Section 4.3. Upon a `put` request, this block store creates meta-data for the block being stored, initializing its replica list to the empty list; if the block being stored already exists, existing meta-data is instead re-used and its reference count is incremented. Figure 34 sketches this process: input files are chopped and indexed, which yields a set of data blocks that are named and stored, while meta-data listing replicas for each of these blocks are created or updated.

The storage costs associated with this approach lie (i) in the fact that data blocks are stored in a block store even though they might still be accessible in the input files and (ii) per-block meta-data is created. The latter is a secondary concern: typically less than 100 bytes of meta-data are stored for each block, and these can be reclaimed once the block has been sufficiently replicated (since this information has become useless).

The former effectively consumes as much storage space as the sum of input files. Duplicating data locally allows the recovery of data from transient failures (e.g., accidental deletion of a file) to be carried out locally, which can be a useful feature. From an engineering viewpoint, storage overhead appears to be less of concern as storage becomes cheaper. Recent version control systems such as Git [Hamano 2006], for instance, achieve better bandwidth-efficiency by systematically storing complete repositories (i.e., the complete revision history of a set of files) locally.

Peer-to-peer file sharing systems such as GNUnet [Bennett *et al.* 2002] sometimes allow users to avoid this overhead by simply storing file names and performing file chopping and indexing *on-demand*. However, this approach shifts the indexing burden to contributor-encounter time, which sounds risky in a context where encounters are unpre-

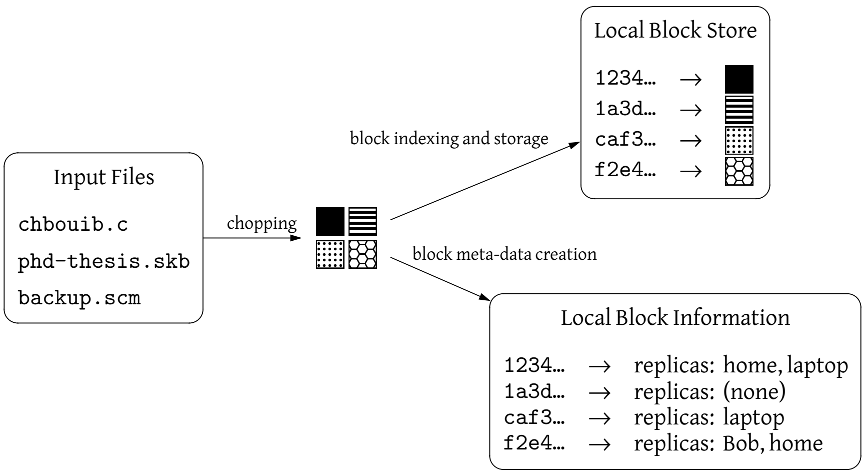


Figure 34. Queueing data blocks for replication.

dictable and short-lived; it is also wasteful CPU-wise since indexing has to be done every time a contributor is encountered.

More importantly, such on-demand indexing is fragile when applied to mutable data: a file could be modified before all the blocks of its previous version have been replicated, and the replication algorithm would have no option but to replicate the *new* blocks, leaving the previous version partially replicated. Figure 35 illustrates the problem: two versions of a given file are created and chopped, one of which has root block R_0 while the next one has root block R_1. The right-hand side of the figure shows two possible replication strategies, labeled A and B: A is based on on-demand indexing, while B is based on pre-indexing and replicates each version entirely. At some point in time, represented by the vertical dashed line, the input file is modified, yielding the version rooted at R_1. The pre-indexing strategy (B) succeeds in replicating the first file version entirely in 7 steps, whereas on-demand indexing (A) fails to complete the replication of any of the two versions in the same amount of time. Thus, should the device crash at that time, the user would be unable to recover *any* version of the file. Overall, we believe the benefits of pre-indexing and local duplication of data blocks are worth the additional storage cost.

This analysis prompts another observation: replication strategies also have to make a tradeoff between *data freshness* and *data availability*. Figure 36 compares pre-indexing and on-demand indexing in a situation where both versions of the file at hand noticeably overlap. While one-demand indexing (A) succeeds (by chance) in replicating the latest version, pre-indexing (B) “only” replicates the old version in the same amount of time. Nevertheless, pre-indexing and local duplication of data blocks allow the implementation of both strategies. A carefully crafted replication algorithm on top of pre-indexing may be able to choose the best strategy in both situations, e.g., conservatively choosing strategy B when there is little overlap between versions (as in Figure 35) and privileging data freshness by

data freshness

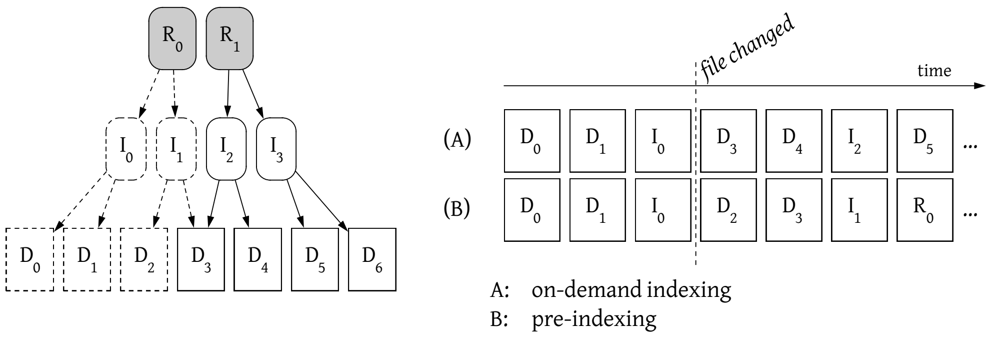


Figure 35. The partial replication problem with mutable data.

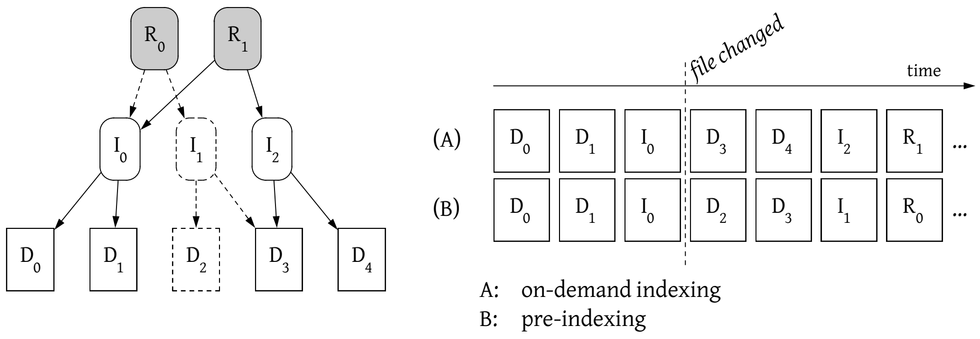


Figure 36. Tradeoff between data freshness and availability.

choosing strategy A when there is noticeable overlap (as in Figure 36). We have not yet designed and implemented such an algorithm, though.

6.3.2. Opportunistic Replication

We consider two aspects of opportunistic data replication: *replication strategies* and related algorithms, as well as *contributor discovery*.

6.3.2.1. Replication Strategies

replication strategies

With the block queuing mechanism we just described, opportunistic replication is essentially a matter of traversing the list of block meta-data structures and selecting blocks for replication upon contributor encounter. Several *replication strategies* that use the available per-block information can be devised. First, per-block replication decisions are easily expressed as a function of a block's meta-data and the name of the device (OpenPGP fingerprint) whose contribution is being considered. One simple, obvious per-block replication strategy is: do not send a block to a contributor if the contributor is known to already hold a copy of the block. Such a strategy can be augmented, for example, as follows: store at least *n* different replicas of the block at *n different* contributors.

In addition, per-device strategies can be defined to decide whether to entrust a particular device with copies of the owner's data blocks. Such strategies are expressed by a function of the device name that returns a boolean indicating whether the device should be accepted as a contributor. This function will usually rely on additional information associated with the device name, such as records of past interactions with that device, as suggested in Chapter 5. An obvious strategy would be to reject contributions of devices that failed to honor their storage commitments in the past. Another strategy is to avoid significant imbalance between the amount of data sent to a contributor and the amount of data stored on its behalf, so that the contributor has an incentive to behave correctly. A smart strategy could adjust the number of replicas needed for a block as a function of the level of trust in each contributor holding a replica. For instance, if two replicas have been distributed but the contributors holding them are strangers, it may be worth distributing at least one additional replica.

Our implementation allows both per-block and per-contributor replication strategies to be defined. The chosen functional programming language lends itself well to the flexibility required by the variety of strategies we want to support: users can supply arbitrary procedures (actually, closures [Saltzer 1978]) to be used as the per-block or per-device strategy. Figure 37 shows Scheme functions implementing a per-block and a per-contributor replication predicate: the former ensures that any given block remains candidate for replication until it has been replicated at least twice, while the latter makes sure that a candidate block is not sent to a peer that is known to already hold a replica of this block[4].

The resulting replication algorithm, parameterized by the two aforementioned predicates, is shown in Figure 38. We assume a `block-sent` primitive that sends the given block to the given remote store, and returns updated block information, i.e., augmenting its list of replicas if replication succeeds. `blocks-replicated-to-server` iterates over a list of blocks, using the predicates to select blocks candidate for replication and sending them. It returns an updated list of blocks: the list of replicas of blocks actually replicated is augmented, while others are left unchanged. Finally, `replicated-blocks` iterates the above process over a list of remote stores and returns an updated list of blocks[5]. This algorithm is $O(nm)$, where n is the number of blocks and m the number of contributors.

[4] In Scheme [Kelsey *et al.* 1998], `let` introduces local bindings. For instance, `(let ((a 2)) body...)` binds variable `a` to value 2 in the following body. Also, `member` is a standard procedure that returns true when its first argument is present in its second argument (a list).

[5] Note that `map` is a standard procedure that applies its first argument (a procedure) to each element of its second argument (a list), and returns the list of results [Kelsey *et al.* 1998]. Likewise, `fold` is the fundamental list iterator, which invokes its first argument (a two-argument procedure) for each element of its third argument [Shivers 1999].

```
(define (candidate-block? block)
  ;; A per-block replication predicate: replicate BLOCK only if it has been
  ;; replicated less than two times.
  (let ((replicas (local-block-info:replicas block)))
    (<= (length replicas) 2)))

(define (candidate-block-for-peer? block peer-fpr)
  ;; A per-contributor replication predicate: replicate BLOCK to the peer
  ;; whose fingerprint is PEER-FPR if the peer does not already hold one
  ;; replica.
  (let ((replicas (local-block-info:replicas block)))
    (not (member peer-fpr replicas))))
```

Figure 37. Example per-block replication strategy.

```
(define (block-sent remote-store block)
  ;; Send BLOCK to REMOTE-STORE.  Return an updated block-info denoting the
  ;; new replication status of BLOCK, or return BLOCK as is if it could not
  ;; be sent.
  …)

(define (blocks-replicated-to-server remote-store blocks)
  ;; Replication BLOCKS to REMOTE-STORE and return a list of block-info
  ;; denoting the new replication status of BLOCKS.
  (map (lambda (block)
         (if (and (candidate-block? block)
                  (let ((fpr (store-fingerprint remote-store)))
                    (candidate-block-for-peer? block fpr)))
             (block-sent block remote-store)
             block))
       blocks))

(define (replicated-blocks remote-stores blocks)
  ;; Iterate over REMOTE-STORES, replicating candidate blocks among BLOCKS to
  ;; selected stores.  Return a list of block-info denoting the new
  ;; replication status of BLOCKS.
  (fold blocks-replicated-to-server
        blocks
        remote-stores))
```

Figure 38. Simplified view of the replication algorithm.

As an optimization, `replicated-blocks` is passed the list of blocks already satisfying `candidate-block?`, and said list of blocks is cached in core memory.

6.3.2.2. Contributor Discovery

storage providers

Our backup daemon treats all *storage providers* (i.e., contributors or other entities implementing the block-store RPC interface seen in Section 4.3.3) in the same way. It can be provided with a static list of IP addresses of storage providers, which allows it to connect to remote storage providers on the network when it is connected to a network infrastructure. In particular, the Internet-based block store that ultimately receives data blocks from contributors and data owners can be specified this way. The backup daemon periodically attempts to connect to these known storage providers. When the network environment makes it possible (e.g., when connected to an infrastructure), it replicates data to them.

For storage providers that are fully trusted by the data owner (e.g., the Internet-based store, the owner's store at their office or home computer), owners may want to specify a suitable replication strategy that stops replicating blocks that have been sent to one such storage provider. This allows the Internet-based store to be handled in an elegant way, without requiring explicit support: only the replication needs to be tailored to take advantage of the fact that it is trusted, while the replication mechanisms remain oblivious to the special-casing of the Internet-based store.

Contributors can also be discovered *dynamically*, e.g., in the neighborhood of a device while it is in *ad hoc* mode. To that end, our implementation uses DNS-SD-over-mDNS as the *service discovery protocol* (SDP) [Cheshire & Krochmal 2006a], specifically the Guile/Scheme programming interface of the Avahi library [Poettering 2006, Courtès 2007]. The protocol was primarily designed for decentralized service discovery among desktop computers. No central server is required, making it suitable to *ad hoc* networking and particularly easy to deploy. However, DNS-SD was not designed with the constraints of mobile devices and wireless *ad hoc* networks in mind and may be less bandwidth- and energy-efficient than SDPs specifically designed for MANETs [Sailhan & Issarny 2005, The UbiSec Project 2005, Helmy 2004, Poettering 2007, Kim *et al.* 2005]. Nevertheless, experimental measurements suggest that mDNS is relatively accurate and energy-efficient, and small improvements were identified that would make it even more suitable to the mobile context [Campo & García-Rubio 2006]. Given that few MANET-specific SDP implementations are available, we considered DNS-SD/mDNS a good choice for practical service discovery in MoSAIC.

service discovery protocol

Concretely, Avahi implements DNS-SD (*DNS Service Discovery*) over mDNS (*Multicast DNS*), a decentralized version of DNS. As the name implies, mDNS is an extension to DNS (the *Domain Name System*) that allows DNS queries to be sent over IP multicast, rather than through unicast communications with a specific server [Cheshire & Krochmal 2006b]. Multicast DNS clients (known as *queriers*) send DNS queries to a given multicast IP address, where each query may contain multiple questions. Clients may also perform *continuous monitoring* (e.g., to provide users with an accurate list of all currently available printers) by sending queries at an exponentially-decreasing rate[6].

Multicast DNS "servers" (known as *responders*) answer *via* multicast[7] with records for which they are authoritative. Answers contain a *time to live* (TTL) indicating for how long the answer is valid. Clients are expected to re-issue queries at 80% of the answer's lifetime if they still have interest in it. Upon startup and subsequent connectivity changes, responders may make an initial announcement of records that are responsible for (by sending a

[6] Multicast DNS makes provisions to limit bandwidth consumption. For instance, when a querier sends a query to which it already knows some answers, it indicates these answers as part of its query, which allows responders to just send information that suffice to update the querier's cache.

[7] That responders answer via multicast allows passive observers (queriers) to update their cache.

gratuitous DNS answer), and they must detect and resolve collisions with names for which they are responsible, using the specified collision resolution protocol. Optionally, responders may send “goodbye packets” to explicitly inform clients of the unavailability of a resource. Finally, mDNS describes a power management framework whereby a *Sleep Proxy Service* answers queries on the behalf of a device that entered a low-power mode, optionally waking up the device when a client requests an active connection to it.

DNS-SD, on the other hand, can be seen essentially as a “convention for naming and structuring DNS resource records” [Cheshire & Krochmal 2006a]. It provides a hierarchical naming scheme for *service types*, akin to the DNS naming scheme. For example, `_http._tcp` designates the “web server” service type, which operates over TCP, while `_workstation._tcp` designates the pseudo-service “workstation”, which allows users to browse neighboring devices.

service types

Avahi contains a system-wide mDNS querier/responder that sends queries and answers on behalf of client applications, which provides a system-wide cache and allows for query aggregation, thereby improving bandwidth efficiency, as noted in [Cheshire & Krochmal 2006b]. It provides an easy-to-use service browsing and resolution programming interface, along with a service publication interface, which relieves applications from the details of the protocol. For instance, applications can create a *service browser* for a given service type; during the browser’s lifetime, an application-provided call-back is invoked any time a new service is discovered or a formerly advertised service has disappeared.

Our backup daemon advertises its presence through Avahi, using the `_block-server._tcp` service type. Beside the IP address and port where the service can be used, advertisements may include the daemon’s OpenPGP fingerprint (in additional ‘TXT’ records, in DNS terminology). When they have pending data blocks to replicate, backup daemons react to advertisements by selecting a subset of the available devices and by replicating data blocks to them. The ability to advertise the OpenPGP fingerprint of participating devices allows backup daemons to select contributing devices without having to connect to them, i.e., without having to go through a TLS handshake with them, which optimizes CPU and network usage. Again, users can provide the daemon with a predicate indicating whether a given device should be rejected as a contributor.

Finally, block replication is also systematically triggered when new data blocks are available for replication, i.e., when a file indexing and block queuing process finishes. In that case, it tries to replicate data blocks to all available storage providers, both statically registered and dynamically discovered.

6.3.2.3. Owner Data Replication

One of our design goals, outlined in Section 2.3, is to have contributors replicate data blocks stored on behalf of data owners to an Internet store. As mentioned earlier, our backup daemon does not differentiate between Internet-based storage providers and contributors. Thus, as for the statically registered storage providers, our backup daemon needs to periodically probe the Internet-based store and, when it is reachable, send it data blocks stored on behalf of data owners that have accumulated. Replicating owner data to an Internet store consists in traversing the list of per-owner stores (which will be discussed in Section 6.3.4), sending each data block they contain and removing them afterwards (since they can be considered “safe”, as seen in Chapter 3).

However, to date, we have not implemented this functionality. The consequence is that data can only be recovered directly from the contributors that received data blocks in the first place. This is a serious shortcoming for many mobile scenarios, which we hope to fix soon. Nevertheless, the data recovery mechanisms discussed later in Section 6.3.3 are suitable, no matter whether the above replication scheme is implemented or not.

6.3.2.4. Adapting to the Network Connectivity

Practical mobile peer-to-peer computing raises the question of control over the available networking devices. While the peer-to-peer service (in our case, the backup daemon) may seek to achieve certain goals in terms of connectivity, device users, on the other hand, may have different, conflicting goals. For instance, the backup daemon would naturally try to maximize contributor encounters, regardless of the connectivity. At the same time, the user may want to use their device for unrelated purposes, e.g., accessing a web site. When, e.g., Wi-Fi is being used, situations could arise where the backup daemon gets the most benefit from using the Wi-Fi interface in *ad hoc* mode, while the user would rather use it in infrastructure mode to connect to the wireless local-area network.

Should the backup daemon or should the end-user have precedence over the networking device usage? There is no single answer to this question. Most likely, some form of resource multiplexing could help both parties share the underlying networking devices. MultiNet/VirtualWifi offers one possible solution by providing a network card driver that virtualizes the underlying network card, thereby allowing simultaneous connections to several Wi-Fi networks [Chandra *et al.* 2004]. The Penumbra broadcast-based network proposes another approach whereby traffic dedicated to peer-to-peer transactions is broadcasted in a device’s neighborhood, while still allowing for regular Wi-Fi traffic *at the same time* [Green 2007].

To some extent, networking device multiplexing remains an open issue. We did not address it in the first iteration of our cooperative backup software. Instead, we let users choose the networking mode of their networking card. The backup daemon remains oblivious to the networking connectivity, and adapts to the current network conditions like any other regular IP-based application.

6.3.3. Data Retrieval

Upon recovery, recovery applications request a series of blocks through `get` RPCs (Section 4.3.3). The first block they request is the one containing the root index and stored under a fixed name, as seen in Section 6.2.3. Once it has been retrieved, the revision pointed to by this block is retrieved (Section 6.2.2). This allows users to know the date of the snapshot they are about to recover from. Although our recovery application does not currently permit it, it would be possible to allows users to choose whether or not wait for a more recent revision.

Data retrieval is implemented using the local block store discussed earlier as a cache. Of course, upon recovery from a catastrophic failure, the local block store will be empty. When a data block is requested by a data recovery tool through a `get` request, it first queries the local block store to see if a copy is available locally. If it's not, it then queries backup daemons at neighboring devices or statically registered storage providers by forwarding them the request as a `get` RPC; if the request succeeds, it populates the local block store for faster access upon future reference. Thus, blocks that are referenced more than once to restore a file system hierarchy (e.g., due to single-instance storage) are made available more quickly.

Note that this corresponds to the general case of data recovery from several storage providers, in *ad hoc* mode. Our main scenario, where data owners recover their data by querying a single store reachable on the Internet (Section 2.3), can be thought of as a special case: provided the Internet-based store uses the same protocol as contributors, it just needs to be specified explicitly to the backup daemon and will automatically be queried upon recovery. Also, remember that, to be able to send `get` requests to storage providers, the backup daemon must have access to the key pair that was used at backup time (Section 5.3.2).

Because data retrieval requires access to data structures handled by the backup daemon (e.g., the local block store and the static storage provider list), this functionality is implemented in the daemon itself, thereby avoiding concurrent access to these data. The daemon exposes it to recovery tools by listening to regular block-store RPCs on a named Unix-domain socket (raw RPCs are used, i.e., without TLS). Therefore, recovery tools can benefit from the daemon-implemented block caching and contributor discovery without having to duplicate it.

In addition, the daemon caches on a per-client basis TLS connections to remote daemons that were opened as a result of a cache miss. The list of cached connections is ordered by storage provider success rate: storage providers that successfully answered the most `get` requests appear first. This allows series of `get` requests not satisfied by the local cache to be rapidly forwarded to storage providers that have succeeded in the past, thereby making full data recovery faster.

We have implemented a small set of client recovery tools. The first one, called `ars-ls`, shows the contents of the directory pointed by the last revision. `ars-retrieve` retrieves the latest available version of the given file or the exact specified revision. Finally, `ars-lookup` traverses the revision chain, showing the index and modification time of a given file every time a directory that introduces a new version of the file is encountered[8]. These clients all use the daemon-implemented recovery RPC interface.

6.3.4. Storage Provision

Backup daemons also listen for TLS connections over TCP from other daemons and serve `put` and `get` RPCs. Daemons can thus act as contributors, handling storage and retrieval requests from their peers. As discussed earlier in Section 5.2, contributors implement *per-owner name spaces*, so that the blocks names of different data owners do not collide. This is achieved by locally maintaining per-owner stores designated by the owner's OpenPGP fingerprint. Upon a successful TLS handshake, the backup daemon opens the peer's block store, and subsequent `put` and `get` RPCs operate on this store.

per-owner name spaces

A contributor can choose to reject storage requests from a data owner, for instance because it does not have sufficient resources, or because of specific knowledge it has about the data owner. The latter can only be done after a successful TLS handshake, when the contributor has the OpenPGP key of the data owner. Again, users can provide the backup daemon with a predicate telling whether a request coming from a particular device should be honored. Such predicates can make use of different kinds of information, allowing a wide range of *cooperation policies* to be defined.

cooperation policies

Among those mentioned in Section 5.3.5, the simplest kind of cooperation policy that may be implemented would be some form of *white list* or *black list*: the cooperation predicate just needs to check whether the device's OpenPGP key is part of one of these lists. More sophisticated policies seeking to allow cooperation with personal acquaintances or "friends of friends", as in [Hubaux *et al.* 2001], are also easily implementable in this framework: such an implementation would traverse the certificate graph rooted at the requesting peer's OpenPGP certificate until it finds a signature of its own.

Cooperation predicates may also use information such as the amount of data already stored on behalf of the requesting device. This allows, for instance, the implementation of a policy that only dedicates small amounts of storage to strangers. A contributor could implement a symmetric trading policy by having its cooperation predicate return true only if the amount of data stored by the requesting peer on its behalf is comparable to the amount of data already stored on behalf of the requesting peer[9]. Note that the success of such a policy would be limited to scenarios where the set of relationships among participating devices is relatively stable.

Another useful source of information that may drive the cooperation decision process is data describing the resources available on the device. Excess-based resource allocation [Grothoff 2003] is implemented by a cooperation predicate that follows the following rules:

[8] From a programming viewpoint, revision chains are conveniently exposed as *streams*, a lazy list-like data structure.

[9] Our implementation currently makes it hard to obtain the amount of personal data stored by a given contributor since the whole list of local block meta-data (Section 6.3.1) needs to be traversed. Future versions may either explicitly store that information or use a relational database that would allow the list of block meta-data to be queried according to various criteria.

1. if large amounts of storage and energy are available, honor storage requests, up to a certain threshold;
2. in times of resource shortage, honor requests based on past behavior, e.g., storing data only on behalf of nodes known to be well-behaved, good contributors.

In the context of mobile devices, all practical cooperation predicates would need to take local resources into account in a similar way.

As mentioned in Chapter 5, cooperation policies that make use of past records of behaviors could be devised. These would require an accurate observation of contributor behavior. For instance, based on local knowledge, each backup daemon could maintain a "trust level" for each encountered contributor. In the retrieval process (Section 6.3.3), successful `get` RPCs could lead the daemon to increase its trust level in the contributor. However, we have not implemented such cooperation policies so far.

6.4. Experimental Measurements

This section describes preliminary measurements conducted on our current implementation of the backup daemon. We first present our experimental setup and methodology and then discuss the results.

6.4.1. Experimental Setup

The following experiments were run using our current version of the backup daemon and related tools on a PC equipped with a 1.8 GHz Intel Pentium M processor and 1 GB of RAM. The backup daemon uses our latest version of *libchop*. In addition, the following libraries and tools were used:

- GNU Guile 1.8.2;
- Avahi 0.6.20 and Guile-Avahi 0.2;
- GnuTLS 2.0.0;
- GNU Guile-RPC 0.2.

Obviously, some of the performance characteristics measured will be highly dependent on the performance of third-party tools that we use, most notably the Scheme interpreter and the ONC RPC and TLS implementations. Likewise, there is a lot of room for optimization in our backup daemon at this stage. Nevertheless, the following measurements should provide useful preliminary insight into the system's performance.

It is important to note that the `put` RPC is implemented as "one-way" RPC in our daemon: the server does not send any reply for this RPC and the client does not have to wait for a reply. This saves the round-trip time that characterizes "regular" RPCs, and allows for batching of RPC calls, as noted in Section 7.4.1 [Srinivasan 1995]. In addition, for the experiments described below, our backup daemons use RSA OpenPGP keys, which allows them to use the `TLS_RSA_WITH_NULL_MD5` cipher suite for communications [Dierks *et al.* 2006], i.e., an encryption-less cipher suite, as suggested in Section 5.4.

We use the *libchop* storage pipeline for file indexing, using a tree stream indexer and a CHK block indexer (see Section 4.3), using Blowfish as the encryption algorithm and SHA-1 hashes as encryption keys and block IDs. All the experiments use a fixed-size chopper; they do not use any lossless compression filter. We use TDB databases [Tridgell *et al.* 1999] for the on-disk storage of the data blocks and block queue.

6.4.2. Indexing and Replication Performance Breakdown

The purpose of this experiment is to provide a breakdown of the time spent by each component of the backup daemon in a typical use case. The experiment works as follows:

1. We start two backup daemons on the same machine without any state, with an empty block store, etc.[10]
2. At some point, usually less than a couple of seconds after startup, backup daemons automatically discover each other (using Avahi).
3. The first backup daemon is instructed to schedule files for backup, using the `ars-command` client, while the second backup daemon is left “idle” (it does not have any file to back up).
4. As soon as its done indexing the files and queuing the resulting blocks (Section 6.3.1), the first backup daemon automatically starts replicating the queued blocks at the second backup daemon (Section 6.3.2).

The file set scheduled for backup is the Ogg Vorbis file set of Section 4.4 (Figure 26), which contains 17 files of 4 MB on average, totalling 69 MB.

The first backup daemon logs cooperation events (storage provider discovery, connection to a storage provider, notification of the end of a data exchange) to a separate *journal* process, using the *journal* RPC interface (Section 6.1). As discussed in Section 6.3.2, replication is normally triggered by three events:

- When new data blocks are available, i.e., after completion of each instance of the file indexing process.
- When a contributor is discovered.

[10] Each backup daemon is provided with an OpenPGP key pair along with ready-to-use TLS Diffie-Hellman (DH) and RSA parameters. Without this, each backup daemon would generate DH and RSA parameters lazily, which takes time.

- Periodically, if statically registered storage providers are reachable.

Therefore, indexing and replication are likely to occur *concurrently*, making it hard to measure the time spent in each on of these two activities.

To allow separate execution time measurement of file indexing on one hand and replication on the other, we slightly modified our backup daemon such that replication is only triggered at the end of the indexing process. We use time stamps associated with the events logged by the backup daemon to measure the time taken to index files as well as the time taken to replicate blocks[11].

Figure 39 shows the performance of the backup daemon with varying block sizes. Several conclusions can be drawn:

- The overall throughput is between 188 KB/s and 436 KB/s, depending on the block size. The replication throughput alone, on the other hand, varies between 264 and 522 KB/s.
- Indexing is between 5 and 2.5 times faster than replication in this context.
- The figure clearly shows per-block overhead, both for indexing and replication. Total execution time grows sub-linearly with respect to the number of blocks[12].

For comparison, copying the same files over SSH[13] takes around 10 seconds, which corresponds to 7 MB/s. We believe several factors justify the relatively poor throughput:

- Our daemon follows a two-stage process, where the results of the first stage (indexing) are written to disk before being fetched back from disk and used by the replication process. The performance of the underlying store (a TDB database in our case) is critical here. Unfortunately, TDB does not maintain an in-memory cache of recently accessed data items, and it writes changes to the database content to disk after each `put` operation[14].
- Although the list of blocks candidate for replication is cached, traversing it certainly incurs some overhead.

[11] Time stamps provided by the journal allow us to measure *real* execution times, as observed by users, as opposed to the time actually spent in the process and/or OS kernel.

[12] The total size of all blocks (not shown here) is almost unchanged as the block size varies, so the overhead is really a function of the number of blocks, not of the total size.

[13] Remember that both the client and server run on the same machine. Note that the SSH protocol provides end-to-end encryption and uses HMACs for communication integrity checks. We used the client and server that come with GNU lsh 2.0.2.

[14] It is possible to instruct TDB not to force writes to disk using the `TDB_NOSYNC` flag, but doing so relaxes transactional semantics, which we want to avoid.

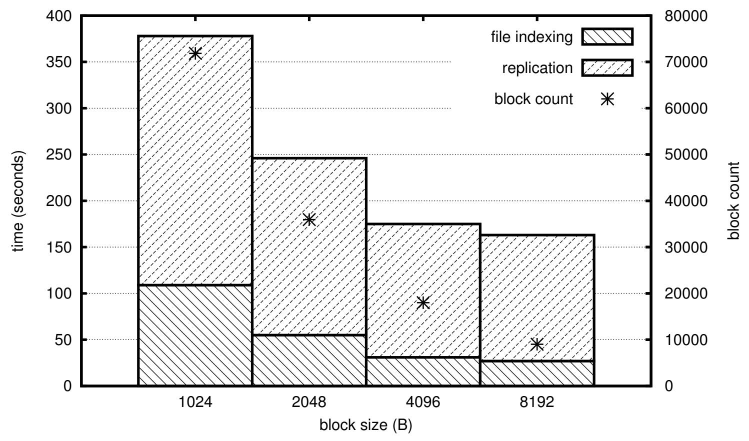


Figure 39. Performance breakdown of the backup daemon with the Ogg Vorbis file set.

- We suspect our ONC RPC implementation to be responsible for a large part of the poor communication performance (it is written in interpreted Scheme code).
- To achieve concurrency of the various activities, our backup daemon uses *coroutines* scheduled cooperatively rather than preemptive multi-threading[15]. While the costs associated with this mechanism are low compared to OS-implemented context switches, repeated switches may affect performance.

Further profiling is needed to confirm these hypotheses.

Figure 40 shows the measurements obtained for the same experiment with the Lout file set (see Figure 26 for details). We observe the similar characteristics as noted earlier with a slightly higher throughput, ranging from 313 kB/s to 572 kB/s. Another observation is that a larger fraction of the total execution time is spent in indexing, which is a direct consequence of the reduction of the number of blocks yielded by single-instance storage.

6.4.3. Data Recovery Performance

The experiment presented here aims to stress-test our data recovery process described in Section 6.3.3, illustrating recovery of data scattered around several storage providers, and to measure its performance. To that end, an experimental environment is set up as follows:

[15] The implementation leverages Scheme closures to realize coroutines. The backup daemon runs its own scheduler that dispatches execution to coroutines and I/O event handlers. Essentially, any routine whose execution may need to be broken up into several pieces (e.g., because its execution would preclude event handling for too long) is written in *continuation-passing style* (this strategy avoids the costs usually associated with `call/cc` [Kelsey *et al.* 1998]). This allows the execution of return continuations to be deferred until some later time.

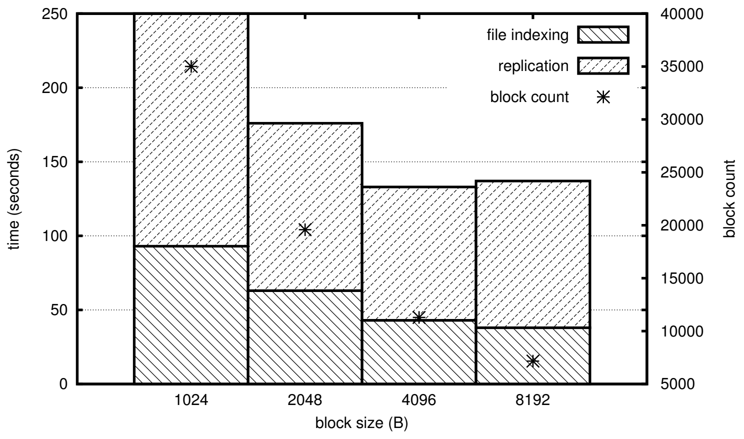


Figure 40. Performance breakdown of the backup daemon with the Lout file set.

1. A fresh environment for the backup daemon that will be used for recovery is created—this daemon plays the role of the *data owner*. Its local block store is empty, as is the case when recovering from a catastrophe.
2. The collection of files we want to recover is chopped and indexed. The resulting blocks are spread over several backup daemons. More precisely, each backup daemon's per-owner database corresponding to the data owner is populated with all the blocks of one file[16]. This simulates a situation where all blocks are replicated once, and where contributors do not fail and are not malicious.
3. Another process then asks the data owner daemon to recover said files, just like the `ars-retrieve` tool does. Since it does not have any of the requested blocks, the daemon is expected to query storage providers it has discovered.

Of course, this experiment allows us to check whether this key functionality actually works. That service discovery works well is critical for this experiment, since failure to find *all* contributors leads to a recovery failure.

For the following measurements, we used the Ogg Vorbis file set. Since it contains 17 files, the experiment spawns 17 contributor daemons and 1 data owner daemon. We ran the experiment with varying block sizes to observe, as earlier, the impact of the block count on performance. In all cases, the data owner was able to recover all 17 files. During the process, it initiates 17 TLS sessions with contributors, which it caches, as described in Section 6.3.3.

[16] Distributing blocks this way simplifies the experiment. Shuffling blocks among daemons would be possible, although it would likely not significantly change the outcome of the experiment.

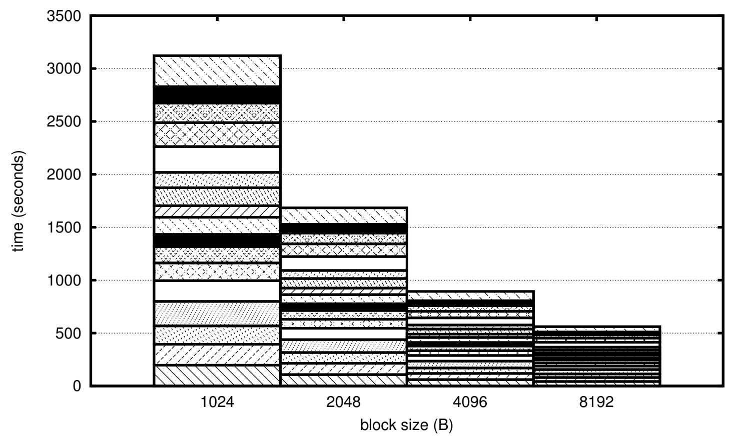


Figure 41. Time to recover files from the Ogg Vorbis file set.

Figure 41 shows the time taken to recover all 17 files as a function of the block size (each slice of the bars represents the time taken to recover one file):

- Again, we observe per-block overhead, with execution time increasing sub-linearly with the number of blocks.
- Throughput varies between 23 KB/s and 127 KB/s, depending on the block size.

Throughput is two orders of magnitude lower than with the SSH client, although the comparison is a bit unfair since the latter performs one-to-one data transfers. Most of the explanations given earlier remain valid here.

In addition, it is worth noting that our current prototype does not batch `get` RPCs. Consequently, both the recovery client and the owner daemon end up fetching blocks one by one, waiting for each `get` RPC to complete; this also magnifies the performance impact of our ONC RPC implementation. Definitely, some form of batching must be implemented to improve performance. Doing so would require changes in *libchop*'s architecture so that tree indexers can take advantage of an `mget` ("multiple get") primitive or similar.

Another possible optimization would be to use IP multicast or similar techniques to send out `get` requests, where possible. This would eliminate the probing period where the daemon sends a `get` request successively to each contributor until one replies successfully, and might even allow for some degree of parallelization. However, TLS sessions would still need to be established with contributors for accountability purposes (Chapter 5), which may limit the performance benefit.

6.5. Summary

The key points brought by this chapter are the following:

- We showed how results and proposals from earlier chapters could be put together in an actual implementation of the cooperative backup service.
- Data structures that complement the storage facilities of Chapter 4 were presented.
- Algorithms used for opportunistic replication, data retrieval, and storage contribution were detailed.
- We discussed the parameterization of these algorithms and proposed actual policies.
- Preliminary experimental results were shown, which allowed us to identify aspects of the implementation and protocol that need to be optimized.

The outcome of this work is a flexible cooperative backup tool that is readily usable in a number of contexts.

Chapter 7. Conclusions and Perspectives

Dependability of mobile devices has become an increasingly important topic. The times where people worked on a single, fixed machine have gone. It has become commonplace to produce data using a variety of mobile devices: PDAs, cell phones, cameras, etc. Ultimately, users expect to be able to "synchronize" data among their devices: data has become mobile as well. Yet, few fault-tolerance mechanisms are available to guarantee the availability of data produced on mobile devices, and existing mechanisms are often restrictive from the end-user viewpoint, for instance because of their reliance on a network infrastructure.

In this dissertation, we tried to address this problem, stating that it should be feasible to improve the dependability of mobile computing devices by leveraging opportunistic cooperation among such devices. To that end, we sketched a *cooperative backup service* for mobile devices, which we call MoSAIC. We explored the design space of such a service in several directions, focusing on issues not addressed by similar services in infrastructure-based contexts. The following section summarizes our approach and contributions. We then discuss research perspectives based on lessons learnt during this effort.

7.1. Summary and Contributions

The design of a cooperative data backup service requires the exploration of a variety of different domains. While we did not explore all the issues that relate to this goal, we tried to explore several different tracks, namely: the dependability of such a service (Chapter 3), distributed storage techniques suitable for data fragmentation-redundancy-scattering (Chapter 4), core mechanisms allowing for secure cooperation among mutually distrustful devices (Chapter 5). We then outlined our implementation of the proposed service that builds on this work (Chapter 6). For each of these domains, our contributions can be summarized as follow.

Analytical dependability evaluation. We proposed a model of the cooperative backup process based on Petri nets and Markov chains, along with a methodology for the dependability evaluation of the cooperative backup service. We identified scenarios where our cooperative backup scheme is beneficial, and showed that MoSAIC can decrease the *probability of data loss* by a factor up to the *ad hoc* to Internet connectivity ratio. A comparison of replication techniques based on *erasure codes* and simple replication techniques, where each data item is replicated in its entirety, showed that erasure coding improves data dependability only in very narrow scenarios. To our knowledge, while similar techniques have been used in the Internet-based peer-to-peer context, no such evaluation had been carried our in a mobile data context.

Distributed storage techniques. We identified key requirements of the storage layer of our cooperative backup service, namely: storage efficiency, scattering of small data blocks, backup atomicity, protection against accidental and malicious data modifications, encryption, and backup redundancy. We analyzed techniques described in the literature that meet these requirements. We described our flexible storage layer implementation, named *libchop*, and used it to carry out an experimental evaluation of combinations of several storage techniques. The evaluation focuses on the storage- and CPU-efficiency of each combination. We concluded on the most suitable combination of storage techniques for our use cases. Our effort differs from earlier work on the evaluation of storage techniques in that it is constrained by the mobile context. In particular, network partitioning and the unpredictable connectivity that characterize it must be addressed.

Secure cooperation. The service we proposed is one in which anyone is free to participate, without having any prior trust relationships with other participants. Threats to cooperation and security were identified. Accountability was suggested as a key requirement to address them. We proposed a set of self-organized, policy neutral security mechanisms, based on the use of self-managed, unique and verifiable designators for devices. We showed how selected self-organized cooperation policies from the literature, such as reputation systems, could be implemented on top of these mechanisms. Using self-managed designators makes the service vulnerable to Sybil attacks; consequently, we studied the impact of such attacks in our contexts and showed how cooperation policies can be implemented to reduce their impact. We outlined an implementation of these core mechanisms using standard protocols, namely TLS with OpenPGP-based authentication. Our approach differs from related work on self-organized cooperative systems in that it focuses on low-level policy-neutral primitives rather than on cooperation policies.

Implementation. Finally, our implementation of the cooperative backup service was depicted, building upon the results and proposals of earlier chapters. Additional storage facilities complementing those studied earlier were described. We described the algorithms and techniques used for opportunistic replication, data retrieval and storage contribution, and discussed their parameterization. The cooperative backup is readily usable in a number of contexts, not limited to mobile devices[1]. Preliminary performance measurements were shown, which allowed us to identify aspects of the implementation and protocol requiring optimization. The end result is a practical cooperative backup tool readily usable in several different contexts, including the mobile context.

[1] For instance, it may be used on a local area network, as well as over the Internet, with a few limitations, though (e.g., service discovery is limited to LANs or wireless LANs and networking technicalities such as NAT traversal are not implemented).

7.2. Lessons Learnt and Perspectives

Although different aspects of the design of the cooperative backup service have been considered, one cannot help thinking about other aspects that have been insufficiently covered, or about further exploring each particular aspect. In the sequel, we try to provide insight into possible research directions that would complement our work.

Dependability evaluation. Our evaluation work could be augmented in several ways. First, replication strategies based on so-called *rate-less* erasure codes could be modeled and evaluated. Rate-less erasure codes allow the production of an unlimited number of distinct fragments, out of which any k suffice to recover the original data [Mitzenmacher 2004, Morcos *et al.* 2006]. Their use could also be evaluated with little impact on our model, for instance rate-less codes could be approximated by choosing higher values of parameter n.

Second, the *worst case* time to backup (e.g., the 90th percentile of the time-to-backup probability distribution) could also be evaluated. This would be a useful metric in cases where the data being backed up can easily be recreated by the user: it allows users to know whether it is worthwhile waiting for the backup or whether they would be better off recreating the data. It would also permit the estimation of the global storage usage for a given data production rate.

Storage techniques. Future work on the optimization of the storage layer may concern several aspects. First, the energy costs of the various design options need to be assessed, especially those related to the wireless transmission of backup data between nodes. The hardware-level approach to measurements of the data compression and transmission energetic costs taken in [Barr & Asanovic 2006] looks promising, although hard to setup and very device- and implementation-dependent. An alternative, higher-level approach could devise and use a model of the energy consumption of the CPU/memory and networking interface. Second, we suggested in Section 4.2.6 that erasure coding could be used both as a chopping algorithm and redundancy technique on input streams themselves rather than on data blocks, presumably with a positive impact on data dependability. Such an approach needs to be implemented and evaluated as well. The latter can be achieved by extending our Petri net model from Chapter 3. Third, it seems likely that no single configuration of the backup service will be appropriate for all situations, so dynamic adaptation of the service to suit different contexts needs to be investigated.

We mainly evaluated compression techniques that operate on arbitrary octet streams. Compact encoding of *structured* data, on the other hand, is a promising approach, as demonstrated by the XMill XML compressor [Liefke & Suciu 2000] and application-specific compact encodings such as [Eisler 2006]. Using knowledge of the data structures they manipulate, such approaches have more opportunities for compression. Furthermore, compact encodings are often already provided by applications or run-time systems themselves [Bobrow & Clark 1979]. Therefore, better cooperation with applications and run-time systems would help improve support for data backup. The large body of work on automated data serialization and on persistent systems provides practical ways to achieve this.

Secure cooperation. Chapter 5 proposed core mechanisms to help achieve accountability in a self-organized fashion. Such mechanisms are a basic requirement and we showed that higher-level cooperation policies could be built on top of them. Simple cooperation policies can be readily implemented in the framework of our backup daemon. More complex strategies such as a reputation system would need additional support, most notably a protocol to exchange reputation data among participants that use the policy. However, we did not attempt to implement and evaluate cooperation policies themselves.

There has been a large amount of work on the evaluation of cooperation policies. Most of them were done through simulation, although some of them were done through large-scale software deployment, notably in the peer-to-peer area [Locher *et al.* 2006]. We chose not to impose any cooperation policy. Thus, an evaluation would need to account of the coexistence of several, potentially significantly different cooperation policies. To our knowledge, little work has been done in this direction.

Implementation. We implemented a cooperative backup daemon that provides the major functionalities we had in mind: opportunistic data replication upon contributor discovery, and data retrieval from available contributors. One missing piece is replication (e.g., to an Internet-based store) of data blocks stored on behalf of data owners. The algorithms we implemented can be easily parameterized, e.g., by providing custom replication predicates. So far, we did not make full use of the flexibility offered. Future work could consist in implementing and evaluating various replication strategies and cooperation policies.

Our preliminary performance evaluation allowed us to pinpoint performance bottlenecks, which we plan to address. Other evaluations would need to be made, e.g., evaluations focusing on data dependability rather than throughput. However, such experiments appear to be hard to design. Finally, we believe that a good evaluation of the proposals made in this dissertation would be an actual deployment of the software. This would provide better insight into its strengths and weaknesses, from a *practical* viewpoint. Eventually releasing it as Free Software may help, although it would obviously take more than this to actually deploy it.

7.3. A Look Forward

Cooperative backup is not a novel idea in itself. However, it had never been applied to the mobile context, which noticeably constrains its design compared to its peer-to-peer Internet-based counterpart, as we saw. However, there has been significant work on distributed file systems, file sharing, and cooperative caching in the context of mobile *ad hoc* networks. To some extent, this work shares a common goal with our cooperative backup service: making data *accessible* from anywhere, as transparently as possible, and optionally allowing for data sharing among users. Ubiquitous data accessibility, e.g., through spontaneous caching, is an important goal: often, mobile devices are actually used as a *cache*, but users currently often end up synchronizing devices manually, through *ad hoc* methods, depending on the devices, data types, etc. A unified and possibly transparent way to handle this is needed.

At first sight, this goal may seem different from that of (cooperative) backup. However, cooperative backup can also be seen as a generalization of such forms of caching: replicating among the user's own devices effectively provides similar caching and allows for ubiquitous data access—and of course, it also increases data availability by leveraging third-party devices.

We also noticed that a large body of work on mobile distributed file systems or caches, targeting scenarios with intermittent connectivity at best, bear some similarity with version control software, and in particular distributed version control. In effect, *some* form of versioning facilities are usually needed to deal with disconnected operation and the creation of diverging replicas, including the ability to merge changes that were made to replicas. We believe that this convergence of goals calls for a *unification* of the distributed caching, backup and version control approaches. Indeed, as each of these fields matures, it appears to duplicating efforts made in one of the others.

While targeting cooperative data backup for MANETs, we also found ourselves doing work similar to approaches in the networking area, namely delay-tolerant networks (DTNs). Our cooperative backup approach—replicating data items at intermediate nodes so that they eventually reach some Internet-based store—can also be seen as a *communication* and routing problem, as in DTNs. The networking viewpoint offered by the DTN literature is enlightening in that respect. With intermittent connectivity patterns becoming more common, DTNs may reflect a growing need. We believe more replication strategies of our cooperative backup service could be inspired by work on routing in DTNs. Similarly, our work can probably contribute to the state of the art in routing strategies for DTNs.

In hindsight, there are many different ways to look at our cooperative backup approach. Future work would benefit from a higher-level, more unified view of the problems we are trying to solve.

Appendix A. Notes on the Storage Layer Implementation

This appendix highlights key aspects of the *libchop* distributed storage library. We first present its reflective programming interface and then show how various features were implemented by taking advantage of it.

A.1. Reflection and Polymorphic Constructors

introspection

The representation of indices as shown in Figure 43 illustrated the basic *introspection* capabilities built in *libchop*. Basically, *libchop* is implemented according to the object-oriented paradigm where, in addition, classes are *reified* at run-time. This allows programs to look up classes by name[1]. When retrieving data from a serialized index, this greatly facilitates the construction of the necessary retrieval pipeline, as illustrated earlier.

Besides, reflection also allowed for the implementation of *virtual constructors*, sometimes referred to as *factories* or as the *prototype* pattern [Gamma *et al.* 1995]. In practice, virtual constructors allow different but related classes to provide a common constructor, such that all of them can be instantiated by invoking the same generic constructor.

Figure 42 provides an example of how this is can be used in practice. Here, all compressor classes (aka. "zip filter" classes) inherit from the `chop_zip_filter_class_t` class, which itself inherits from the canonical class type, `chop_class_t`. This example shows how to instantiate an arbitrary zip filter from the name of a class: the class is looked up and, if found and proved to be a zip filter class, it is instantiated.

The construction of a retrieval pipeline from the serialized index, as discussed above, builds on this feature. It also enables rapid adaptation to different configurations, as is the case for the set of experiments described in Section 4.4. Unit tests were also easily extended to support all implementations of a given interface, thereby providing good test coverage at little or no cost.

[1] To that end, a perfect hash table that maps class names to class objects is created at compilation-time using GNU Gperf. This makes class lookup $O(1)$, allowing it to be used in a wide range of contexts.

```
const chop_class_t *zip_class;

zip_class = chop_class_lookup ("lzo_zip_filter");
if ((zip_class != NULL) &&
    (chop_object_is_a ((chop_object_t *) zip_class,
                       &chop_zip_filter_class)))
 {
   /* ZIP_CLASS is actually a "zip filter" class so we can use it
to create a new filter.  */
   chop_filter_t *zip_filter;

   /* Allocate room for a ZIP_CLASS instance.  */
   zip_filter = chop_class_alloca_instance (zip_class);

   /* Create an instance of ZIP_CLASS.  */
   err = chop_zip_filter_generic_open ((chop_zip_filter_class_t *) zip_class,
                                      CHOP_ZIP_FILTER_DEFAULT_COMPRESSION,
                                      0, zip_filter);
   if (err)
     exit (1);
   else
     {
       /* Use ZIP_FILTER…  */
     }
 }
else
 {
   /* We couldn't find the requested class or it is not a
zip filter class. */
   exit (2);
 }
```

Figure 42. Example use of introspection and polymorphic constructors in *libchop*.

A.2. Stand-Alone Tools

Our library comes with several stand-alone tools. The main one is `chop-archiver`. It provides a command-line interface to the complete storage and retrieval pipelines. The `-archive` option instructs it to process an input file with the storage pipeline, eventually storing it as a set of blocks in a block store. The `-restore` option instructs it to restore a data stream by fetching and reassembling its constituting blocks.

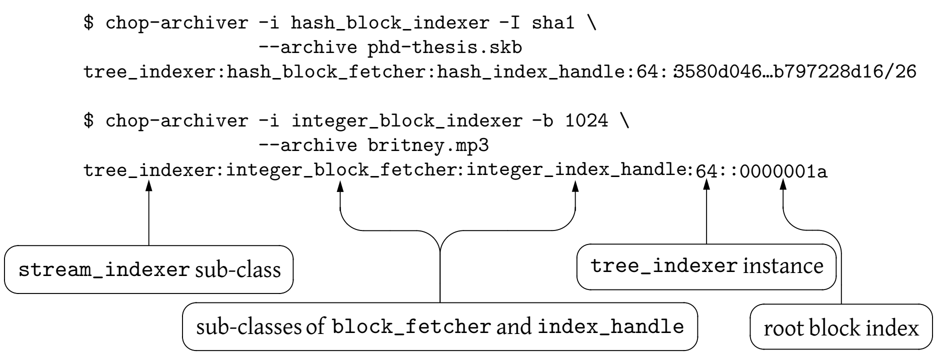
In addition, thanks to the aforementioned reflective capabilities of the libraries, the tool offers command-line options to select the block store, indexer, block indexer and chopper classes to be used (illustrated in Figure 43, see below). It can also be used to access both a local block store or a remote block store, as described in Section 4.3.3.

The `chop-block-server` tool is a server implementing the block store RPC interface. It handles `put` and `get` requests by forwarding them to a local, on-disk store. Note that it is a very simple implementation, as it does not offer all the possibilities of our backup daemon (see Section 6).

A.3. Serialization of Meta-Data Root Indices

As mentioned earlier, the storage process returns an index that may be used to retrieve the stored data. Figure 43 shows serialized forms of such indices as printed by the `chop-archiver` tool: the first invocation uses a `hash` block indexer, while the second uses an `integer` block indexer. Those indices contain all the information necessary to retrieve the data: it lists the indexer, block fetcher and corresponding index classes that should be used for recovery, along with the actual index in serialized form; in the first case, that index is a 160-bit SHA-1 hash whereas in the second case it is a 32-bit integer. Upon retrieval, all the necessary classes can be looked up and instantiated from the serialized forms in the serialized index. This allows the creation of a complete suitable retrieval pipeline directly from the serialized index.

When filters are used, e.g., by means of a filtered input stream or filtered block store, neither the filter name nor the mere fact that a filter was used appears in the resulting index. This is because the use of filters is transparent to the storage (and retrieval) pipeline. Thus, users must arrange to use the same filters upon retrieval as were used at storage-time.



```
$ chop-archiver -i hash_block_indexer -I sha1 \
                --archive phd-thesis.skb
tree_indexer:hash_block_fetcher:hash_index_handle:64: 3580d046…b797228d16/26

$ chop-archiver -i integer_block_indexer -b 1024 \
                --archive britney.mp3
tree_indexer:integer_block_fetcher:integer_index_handle:64::0000001a
```

Figure 43. ASCII-serialized index produced by *libchop*'s storage process.

Bibliography

[Aad *et al.* 2006]
I. Aad, C. Castelluccia, J-P. Hubaux. Packet Coding for Strong Anonymity in Ad Hoc Networks. In *Proc. of the nternational Conf. on Security and Privacy in Communication Networks (Securecomm)*, pp. 1–10, August 2006.

[Aiyer *et al.* 2005]
A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J-P. Martin, C. Porth. BAR Fault Tolerance for Cooperative Services. In *Proc. of the ACM Symp. on Operating Systems Principles*, pp. 45–58, October 2005.

[Allmydata, Inc. 2007]
Allmydata, Inc.. Allmydata Tahoe: A Distributed Storage and Backup Tool. 2007. *http://allmydata.org/*.

[Aspelund 2005]
T. Aspelund. Retrivability of Data in Ad-Hoc Backup (Master Thesis). University of Oslo, Norway, May 2005. *http://research.iu.hio.no/theses/master2005/*.

[Avizienis *et al.* 2004]
A. Avizienis, J-C. Laprie, B. Randell, C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. In *IEEE Transactions on Dependable and Secure Computing*, (1)pp. 11–33, IEEE CS Press, 2004.

[Backus 1978]
J. Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. In *Communications of the ACM*, 21(8) , New York, NY, USA, August 1978, pp. 613–641.

[Barr & Asanovic 2006]
K. Barr, K. Asanovic. Energy Aware Lossless Data Compression. In *ACM Transactions on Computer Systems*, 24(3) , August 2006, pp. 250–291.

[Barreto & Ferreira 2004]
J. Barreto, P. Ferreira. A Replicated File System for Resource Constrained Mobile Devices. In *Proc. of the IADIS Int. Conf. on Applied Computing*, March 2004.

[Batten *et al.* 2001]
C. Batten, K. Barr, A. Saraf, S. Treptin. pStore: A Secure Peer-to-Peer Backup System. Technical Report MIT-LCS-TM-632, MIT Laboratory for Computer Science, December 2001.

[Bennett *et al.* 2002]
K. Bennett, C. Grothoff, T. Horozov, I. Patrascu. Efficient Sharing of Encrypted Data. In *Proc. of the 7th Australasian Conf. on Information Security and Privacy (ACISP 2002)*, Lecture Notes in Computer Science, (2384)pp. 107–120, Springer-Verlag, 2002.

[Bennett *et al.* 2003]
K. Bennett, C. Grothoff, T. Horozov, J. T. Lindgren. An Encoding for Censorship-Resistant Sharing. In , , 2003, .

[Bhagwan *et al.* 2004]
R. Bhagwan, K. Tati, Y-C. Cheng, S. Savage, G. M. Voelker. Total Recall: System Support for Automated Availability Management. In *Proc. of the ACM/USENIX Symp. on Networked Systems Design and Implementation*, March 2004.

[Bicket *et al.* 2005]
J. Bicket, D. Aguayo, S. Biswas, R. Morris. Architecture and Evaluation of an Unplanned 802.11b Mesh Network. In *Proc. of the Int. Conf. on Mobile Computing and Networking (MobiCom)*, pp. 31–42, ACM Press, 2005.

[Birrell *et al.* 1987]
A. Birrell, M. B. Jones, T. Wobber. A Simple and Efficient Implementation for Small Databases. In *Proc. of the 11th ACM Symp. on Operating System Principles*, pp. 149–154, ACM Press, November 1987.

[Black 2006]
J. Black. Compare-by-Hash: A Reasoned Analysis. In *Proc. of the USENIX Annual Technical Conf., Systems and Experience Track*, 2006.

[Bobrow & Clark 1979]
D. G. Bobrow, D. W. Clark. Compact Encodings of List Structure. In *ACM Transactions on Programming Languages and Systems*, 1(2) , October 1979, pp. 266–286.

[Bolosky *et al.* 2000]
W. J. Bolosky, J. R. Douceur, D. Ely, M. Theimer. Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs. In *Proc. of the Int. Conf. on Measurement and Modeling of Computer Systems*, pp. 34–43, 2000.

[Boulkenafed & Issarny 2003]
M. Boulkenafed, V. Issarny. AdHocFS: Sharing Files in WLANs. In *Proc. of the 2nd Int. Symp. on Network Computing and Applications*, April 2003.

[Buchegger & Boudec 2003]
S. Buchegger, J-Y. L. Boudec. The Effect of Rumor Spreading in Reputation Systems for Mobile Ad-hoc Networks. In *Proc. of WiOpt '03: Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks*, March 2003.

[Buttyán & Hubaux 2000]
L. Buttyán, J-P. Hubaux. Enforcing Service Availability in Mobile Ad-Hoc WANs. In *Proc. of the 1st ACM Int. Symp. on Mobile Ad Hoc Networking & Computing*, pp. 87–96, IEEE CS Press, 2000.

[Buttyán & Hubaux 2003]
L. Buttyán, J-P. Hubaux. Stimulating Cooperation in Self-Organizing Mobile Ad Hoc Networks. In *ACM/Kluwer Mobile Networks and Applications*, 8(5) , October 2003, pp. 579–592.

[Béounes *et al.* 1993]
C. Béounes, M. Aguéra, J. Arlat, K. Kanoun, J-C. Laprie, S. Metge, S. Bachmann, C. Bourdeau., J-E. Doucet, D. Powell, P. Spiesser. SURF-2: A Program for Dependability Evaluation of Complex Hardware and Software Systems. In *Proc. of the Twenty-3rd IEEE Annual Int. Symp. on Fault-Tolerant Computing*, pp. 668–673, June 1993.

[Callas *et al.* 1998]
J. Callas, L. Donnerhacke, H. Finney, R. Thayer. OpenPGP Message Format (RFC 2440). Internet Engineering Task Force (IETF), November 1998. *http://tools.ietf.org/html/rfc2440.*

[Campo & García-Rubio 2006]
C. Campo, C. García-Rubio. DNS-Based Service Discovery in Ad Hoc Networks: Evaluation and Improvements. In *Proc. of the Int. Conf. on Personal Wireless Communications*, pp. 111–122, Springer-Verlag, September 2006.

[Capkun *et al.* 2002]
S. Capkun, L. Buttyán, J-P. Hubaux. Small Worlds in Security Systems: an Analysis of the PGP Certificate Graph. In *Proc. of the Workshop on New Security Paradigms*, pp. 28–35, ACM Press, 2002.

[Capkun *et al.* 2003]
S. Capkun, L. Buttyán, J-P. Hubaux. Self-Organized Public-Key Management for Mobile Ad Hoc Networks. In *IEEE Transactions on Mobile Computing*, 2(1) , January 2003, pp. 52–64.

[Cerf *et al.* 2007]
V. G. Cerf, S. C. Burleigh, R. C. Durst, K. Fall, A. J. Hooke, K. L. Scott, L. Torgerson, H. S. Weiss. Delay-Tolerant Networking Architecture (RFC 4838). Google Corporation, VA, USA, April 2007. *http://tools.ietf.org/html/rfc4838.*

[Chandra *et al.* 2004]
R. Chandra, P. Bahl, P. Bahl. MultiNet: Connecting to Multiple IEEE 802.11 Networks Using a Single Wireless Card. In *Proc. of the Annual Joint Conf. of the IEEE Computer and Communications Societies*, pp. 882–893, March 2004.

[Cheshire & Krochmal 2006a]
S. Cheshire, M. Krochmal. DNS-Based Service Discovery. Apple Computer, Inc., August 2006. *http://www.dns-sd.org/.*

[Cheshire & Krochmal 2006b]
S. Cheshire, M. Krochmal. Multicast DNS. Apple Computer, Inc., August 2006. *http://www.multicastdns.org/.*

[Chiu 1999]
A. Chiu. Authentication Mechanisms for ONC RPC (RFC 2695). Internet Engineering Task Force (IETF), September 1999. *http://tools.ietf.org/html/rfc2695.*

[Clarke *et al.* 2001]
I. Clarke, O. Sandberg, B. Wiley, T. W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Proc. of the Int. Workshop on Designing Privacy Enhancing Technologies*, pp. 46–66, Springer-Verlag, 2001.

[Clausen & Jacquet 2003]
T. H. Clausen, P. Jacquet. Optimized Link State Routing Protocol (RFC 3626). INRIA Rocquencourt, France, October 2003. *http://tools.ietf.org/html/rfc3626.*

[Cooley *et al.* 2004]
J. Cooley, C. Taylor, A. Peacock. ABS: The Apportioned Backup System. Technical Report , MIT Laboratory for Computer Science 2004.

[Cornell *et al.* 2004]
B. Cornell, P. Dinda, F. Bustamante. Wayback: A User-level Versioning File System for Linux. In *Proc. of the USENIX Annual Technical Conf., FREENIX Track*, pp. 19–28, 2004.

[Cornelli *et al.* 2002]
F. Cornelli, E. Damiani, S. D. C. d. Vimercati, S. Paraboschi, P. Samarati. Choosing Reputable Servents in a P2P Network. In *Proc. of the 11th World Wide Web Conf.*, May 2002.

[Courtès *et al.* 2006]
L. Courtès, M-O. Killijian, D. Powell. Storage Tradeoffs in a Collaborative Backup Service for Mobile Devices. In *Proc. of the 6th European Dependable Computing Conf.*, pp. 129–138, IEEE CS Press, October 2006.

[Courtès *et al.* 2007a]
L. Courtès, O. Hamouda, M. Kaâniche, M-O. Killijian, D. Powell. Dependability Evaluation of Cooperative Backup Strategies for Mobile Devices. In *Proc. of the IEEE Int. Symp. on Pacific Rim Dependable Computing*, December 2007.

[Courtès *et al.* 2007b]
L. Courtès, M-O. Killijian, D. Powell. Security Rationale for a Cooperative Backup Service for Mobile Devices. In *Proc. of the Latin-American Symp. on Dependable Computing*, pp. 212–230, Springer-Verlag, September 2007.

[Courtès 2007]
L. Courtès. Using Avahi in Guile Scheme Programs. June 2007. *http://www.nongnu.org/guile-avahi/.*

[Cox & Miller 1965]
D.R Cox, H.D. Miller. The Theory of Stochastic Processes. Chapman and Hall Ltd., 1965.

[Cox *et al.* 2002]
L. P. Cox, C. D. Murray, B. D. Noble. Pastiche: Making Backup Cheap and Easy. In *5th USENIX Symp. on Operating Systems Design and Implementation*, pp. 285–298, December 2002.

[Cox & Noble 2003]
L. P. Cox, B. D. Noble. Samsara: Honor Among Thieves in Peer-to-Peer Storage. In *Proc. 19th ACM Symp. on Operating Systems Principles*, pp. 120–132, October 2003.

[Dabek *et al.* 2001]
F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica. Wide-Area Cooperative Storage With CFS. In *Proc. 18th ACM Symp. on Operating Systems Principles*, pp. 202–215, October 2001.

[Demers *et al.* 1994]
A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, B. Welch. The Bayou Architecture: Support for Data Sharing Among Mobile Users. In *Proc. of the Workshop on Mobile Computing Systems and Applications*, pp. 2–7, IEEE CS Press, December 1994.

[Deswarte *et al.* 1991]
Y. Deswarte, L. Blain, J-C. Fabre. Intrusion Tolerance in Distributed Computing Systems. In *Proc. of the IEEE Symp. on Research in Security and Privacy*, pp. 110–121, May 1991.

[Deutsch 1996]
P. Deutsch. DEFLATE Compressed Data Format Specification Version 1.3 (RFC 1951). Internet Engineering Task Force (IETF), May 1996. *http://tools.ietf.org/html/rfc1951.html.*

[Deutsch & Gailly 1996]
P. Deutsch, J-L. Gailly. ZLIB Compressed Data Format Specification Version 3.3 (RFC 1950). Internet Engineering Task Force (IETF), May 1996. *http://tools.ietf.org/html/rfc1950.html.*

[Dierks *et al.* 2006]
T. Dierks, E. Rescorla, W. Teerse. The Transport Layer Security (TLS) Protocol, Version 1.1 (RFC 4346). Internet Engineering Task Force (IETF), April 2006. *http://tools.ietf.org/html/rfc4346.*

[Dingledine *et al.* 2001]
R. Dingledine, M. J. Freedman, D. Molnar. Accountability. In *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, O'Reilly & Associates, Inc., Andy Oram (editor), Chapter 16, pp. 271–341, March 2001.

[Dolstra & Hemel 2007]
E. Dolstra, A. Hemel. Purely Functional System Configuration Management. In *Proc. of the Workshop on Hot Topics in Operating Systems (HotOS XI)*, USENIX Association, May 2007.

[Douceur 2002]
J. R. Douceur. The Sybil Attack. In *Revised Papers from the 1st Int. Workshop on Peer-to-Peer Systems (IPTPS)*, pp. 251–260, Springer-Verlag, 2002.

[Eisler *et al.* 1997]
M. Eisler, A. Chiu, L. Ling. RPCSEC_GSS Protocol Specification (RFC 2203). Internet Engineering Task Force (IETF), September 1997. *http://tools.ietf.org/html/rfc2203.*

[Eisler 2006]
M. Eisler. XDR: External Data Representation Standard (RFC 4506). Internet Engineering Task Force (IETF), May 2006. *http://tools.ietf.org/html/rfc4506.html.*

[Ellison 1996]
C. M. Ellison. Establishing Identity Without Certification Authorities. In *Proc. of the 6th USENIX Security Symp.*, pp. 67–76, 1996.

[Ellison *et al.* 1999]
C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, T. Ylonen. SPKI Certificate Theory (RFC 2693). Internet Engineering Task Force (IETF), September 1999. *http://www.ietf.org/rfc/rfc2693.txt.*

[Elnikety *et al.* 2002]
S. Elnikety, M. Lillibridge, M. Burrows. Cooperative Backup System. In *The USENIX Conf. on File and Storage Technologies (Work in Progress Report)*, January 2002.

[Elnozahy 2002]
E. N. Elnozahy. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. In *ACM Computing Surveys*, 34(3) , September 2002, pp. 375–408.

[Ershov 1958]
A. P. Ershov. On Programming of Arithmetic Operations. In *Communications of the ACM*, pp. 3–6, ACM Press, August 1958.

[Fall 2003]
K. Fall. A Delay-Tolerant Network Architecture for Challenged Internets. In *Proc. of the Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pp. 27–34, August 2003.

[Farrell & Cahill 2006]
S. Farrell, V. Cahill. Security Considerations in Space and Delay Tolerant Networks. In *Proc. of the 2nd IEEE Int. Conf. on Space Mission Challenges for Information Technology*, pp. 29–38, IEEE CS Press, 2006.

[Feeney & Nilsson 2001]
L. M. Feeney, M. Nilsson. Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Networking Environment. In *Proc. of the 20th IEEE Conf. on Computer Communications (IEEE InfoCom)*, pp. 1548–1557, April 2001.

[Fischer *et al.* 1985]
M. J. Fischer, N. A. Lynch, M. S. Paterson. Impossibility of Distributed Consensus with one Faulty Process. In *Journal of the ACM*, 32, April 1985, pp. 374–382.

[Flinn *et al.* 2003]
J. Flinn, S. Sinnamohideen, N. Tolia, M. Satyanarayanan. Data Staging on Untrusted Surrogates. In *Proc. of the USENIX Conf. on File and Storage Technologies (FAST)*, March 2003.

[Gamma *et al.* 1995]
E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns—Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[Ganger *et al.* 2001]
G. R. Ganger, P. K. Khosla, M. Bakkaloglu, M. W. Bigrigg, G. R. Goodson, S. Oguz, V. Pandurangan, C. A. N. Soules, J. D. Strunk, J. J. Wylie. Survivable Storage Systems. In *Proc. of the DARPA Information Survivability Conf. & Exposition (DISCEX)*, pp. 184–195, IEEE CS Press, June 2001.

[Gansner *et al.* 1993]
E. R. Gansner, E. Koutsofios, S. C. North, K-P. Vo. A Technique for Drawing Directed Graphs. In *IEEE Transactions on Software Engineering*, 19(3) , March 1993, pp. 214–230.

[Gibson & Miller 1998]
T. J. Gibson, E. L. Miller. Long-Term File Activity Patterns in a UNIX Workstation Environment. In *Proc. of the 15th IEEE Symp. on Mass Storage Systems*, pp. 355–372, March 1998.

[Girao *et al.* 2007]
J. Girao, D. Westhoff, E. Mykletun, T. Araki. TinyPEDS: Tiny Persistent Encrypted Data Storage in Asynchronous Wireless Sensor Networks. In *Ad Hoc Networks*, 5(7) , September 2007, pp. 1073–1089.

[Goland *et al.* 1999]
Y. Y. Goland, T. Cai, P. Leach, Y. Gu, S. Albright. Simple Service Discovery Protocol/1. 0 Operating without an Arbiter. Internet Engineering Task Force (IETF), October 1999. *http://quimby.gnus.org/internet-drafts/draft-cai-ssdp-v1-03.txt.*

[Goldberg & Yianilos 1998]
A. V. Goldberg, P. N. Yianilos. Towards an Archival Intermemory. In *Proc. IEEE Int. Forum on Research and Technology Advances in Digital Libraries (ADL'98)*, pp. 147–156, IEEE Society, April 1998.

[Gray 1981]
J. Gray. The Transaction Concept: Virtues and Limitations. In *Proc. of the Int. Conf. on Very Large Data Bases*, pp. 144–154, IEEE CS Press, September 1981.

[Green 2007]
A. Green. The Penumbra Broadcast-Based Wireless Network. January 2007. *http://penumbra.warmcat.com/.*

[Grothoff 2003]
C. Grothoff. An Excess-Based Economic Model for Resource Allocation in Peer-to-Peer Networks. In *Wirtschaftsinformatik*, 45(3) , June 2003, pp. 285–292.

[Guttman *et al.* 1999]
E. Guttman, C. Perkins, J. Veizades, M. Day. RFC 2608 -- Service Location Protocol, Version 2. Internet Engineering Task Force (IETF), June 1999. *http://tools.ietf.org/html/rfc2608.*

[Hamano 2006]
J. C. Hamano. GIT—A Stupid Content Tracker. In *Proc. of the Linux Symp., Volume One*, pp. 385–393, July 2006.

[Hamouda 2006]
O. Hamouda. Évaluation de la sûreté de fonctionnement d'un système de sauvegarde coopérative pour des dispositifs mobiles (Master de recherche). LAAS-CNRS, Toulouse, France, 2006.

[Hardin 1968]
G. Hardin. The Tragedy of the Commons. In *Science*, (162) , 1968, .

[Harras *et al.* 2007]
K. A. Harras, M. P. Wittie, K. C. Almeroth, E. M. Belding. ParaNets: A Parallel Network Architecture for Challenged Networks. In *Proc. of the IEEE Workshop on Mobile Computing Systems and Applications*, February 2007.

[Helmy 2004]
A. Helmy. Efficient Resource Discovery in Wireless Ad Hoc Networks: Contacts Do Help. In *Resource Management in Wireless Networking*, Kluwer Academic Publishers, May 2004.

[Henson 2003]
V. Henson. An Analysis of Compare-by-hash. In *Proc. of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems*, pp. 13–18, May 2003.

[Heydon *et al.* 2001]
A. Heydon, R. Levin, T. Mann, Y. Yu. The Vesta Approach to Software Configuration Management. Technical Report Compaq Systems Research Center Research Report 168, Compaq Systems Research Center, USA, March 2001.

[Hitz *et al.* 1994]
D. Hitz, J. Lau, M. Malcolm. File System Design for an NFS File Server Appliance. In *Proc. of the USENIX Winter Technical Conf.*, The USENIX Association, January 1994.

[Hubaux *et al.* 2001]
J-P. Hubaux, L. Buttyan, S. Capkun. The Quest for Security in Mobile Ad Hoc Networks. In *Proc. of the 2nd ACM Int. Symp. on Mobile Ad Hoc Networking & Computing*, pp. 146–155, October 2001.

[Hunt *et al.* 1996]
J. J. Hunt, K-P. Vo, W. F. Tichy. An Empirical Study of Delta Algorithms. In *Software configuration management: ICSE 96 SCM-6 Workshop*, pp. 49–66, Springer, 1996.

[IEEE-SA Standards Board 1999]
IEEE-SA Standards Board. ANSI/IEEE Std 802.11---Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. 1999. *http://standards.ieee.org/*.

[Jaffer *et al.* 1996]
A. Jaffer, T. Lord, J. Blandy, M. Vollmer, M. Djurfeldt. GNU Guile: GNU's Ubiquitous Intelligent Language for Extension. GNU Project, 1996. *http://www.gnu.org/software/guile/*.

[Jain & Agrawal 2003]
S. Jain, D. P. Agrawal. Wireless Community Networks. In *IEEE Computer*, 36(8) , August 2003, pp. 90–92.

[Jain *et al.* 2005]
S. Jain, M. Demmer, R. Patra, K. Fall. Using Redundancy to Cope with Failures in a Delay Tolerant Network. In *Proc. of the Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pp. 109–120, ACM Press, 2005.

[Josefsson & Mavrogiannopoulos 2006]
S. Josefsson, N. Mavrogiannopoulos. The GNU TLS Library. 2006. *http://gnutls.org/*.

[Juang *et al.* 2002]
P. Juang, H. Oki, Y. Wang, M. Martonosi, L-S. Peh, D. Rubenstein. Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet. In *ACM SIGOPS Operating Systems Review*, 36, 2002, pp. 96–107.

[Karypidis & Lalis 2005]
A. Karypidis, S. Lalis. Exploiting Co-Location History for Efficient Service Selection in Ubiquitous Computing Systems. In *Proc. of the Int. Conf. on Mobile and Ubiquitous Systems (MobiQuitous)*, pp. 202–209, IEEE CS Press, July 2005.

[Karypidis & Lalis 2006]
A. Karypidis, S. Lalis. OmniStore: A System for Ubiquitous Personal Storage Management. In *Proc. of the Annual IEEE Int. Conf. on Pervasive Computing and Communications (PerCom)*, pp. 136–147, IEEE CS Press, March 2006.

[Kelsey *et al.* 1998]
R. Kelsey, W. Clinger, J. Rees. Revised5 Report on the Algorithmic Language Scheme. In *Higher-Order and Symbolic Computation*, 11(1) , August 1998, pp. 7–105.

[Kemeny & Snell 1960]
J. G. Kemeny, J. L. Snell. Finite Markov Chains. D. Van Nostrand Co., Inc., Princeton, New Jersey, USA, 1960.

[Kim *et al.* 2005]
M. J. Kim, M. Kumar, B. A. Shirazi. Service Discovery using Volunteer Nodes for Pervasive Environments. In *Proc. of the Int. Conf. on Pervasive Services (ICPS)*, pp. 188–197, July 2005.

[Klemm *et al.* 2003]
A. Klemm, C. Lindemann, O. P. Waldhorst. A Special-Purpose Peer-to-Peer File Sharing System for Mobile Ad Hoc Networks. In *Proc. of the 58th Vehicular Technology Conf.*, pp. 2758–2763, October 2003.

[Kubiatowicz *et al.* 2000]
J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proc. of the 9th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, pp. 190–201, November 2000.

[Kulkarni *et al.* 2004]
P. Kulkarni, F. Douglis, J. LaVoie, J. M. Tracey. Redundancy Elimination Within Large Collections of Files. In *Proc. of the USENIX Annual Technical Conf.*, 2004.

[Kügler 2003]
D. Kügler. An Analysis of GNUnet and the Implications for Anonymous, Censorship-Resistant Networks. In *Proc. of the Conf. on Privacy Enhancing Technologies*, Lecture Notes in Computer Science, pp. 161–176, Springer, March 2003.

[Lai *et al.* 2003]
K. Lai, M. Feldman, J. Chuang, I. Stoica. Incentives for Cooperation in Peer-to-Peer Networks. In *Proc. of the Workshop on Economics of Peer-to-Peer Systems*, 2003.

[Landers *et al.* 2004]
M. Landers, H. Zhang, K-L. Tan. PeerStore: Better Performance by Relaxing in Peer-to-Peer Backup. In *Proc. of the 4th Int. Conf. on Peer-to-Peer Computing*, pp. 72–79, August 2004.

[Lee *et al.* 1999]
Y-W. Lee, K-S. Leung, M. Satyanarayanan. Operation-based Update Propagation in a Mobile File System. In *Proc. of the USENIX Annual Technical Conf.*, pp. 43–56, June 1999.

[Liao *et al.* 2006]
Y. Liao, K. Tan, Z. Zhang, L. Gao. Estimation Based Erasure Coding Routing in Delay Tolerant Networks. In *Proc. of the Int. Conf. on Communications and Mobile Computing*, pp. 557–562, ACM Press, 2006.

[Liedtke 1993]
J. Liedtke. A Persistent System in Real Use — Experiences of the First 13 Years. In *Proc. of the Int. Workshop on Object-Orientation in Operating Systems (I-WOOOS)*, pp. 2–11, December 1993.

[Liefke & Suciu 2000]
H. Liefke, D. Suciu. XMill: an Efficient Compressor for XML Data. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pp. 153–164, ACM Press, 2000.

[Lillibridge *et al.* 2003]
M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, M. Isard. A Cooperative Internet Backup Scheme. In *Proc. of the USENIX Annual Technical Conf.*, pp. 29–42, June 2003.

[Lin *et al.* 2004]
W. K. Lin, D. M. Chiu, Y. B. Lee. Erasure Code Replication Revisited. In *Proc. of the 4th Int. Conf. on Peer-to-Peer Computing*, pp. 90–97, 2004.

[Locher *et al.* 2006]
T. Locher, P. Moor, S. Schmid, R. Wattenhofer. Free Riding in BitTorrent is Cheap. In *Proc. of the 5th Workshop on Hot Topics in Networks (HotNets V)*, pp. 85–90, November 2006.

[Loo *et al.* 2003]
B. T. Loo, A. LaMarca, G. Borriello. Peer-To-Peer Backup for Personal Area Networks. Technical Report IRS-TR-02-015, UC Berkeley; Intel Seattle Research (USA), May 2003.

[Lord 2005]
T. Lord. The GNU Arch Distributed Revision Control System. 2005. *http://www.gnu.org/software/gnu-arch/*.

[Manber 1994]
U. Manber. Finding Similar Files in a Large File System. In *Proc. of the USENIX Winter 1994 Conf.*, pp. 1–10, January 1994.

[Marsan *et al.* 1995]
M.A. Marsan, G. Balbo, G. Conte, S. Donatelli, G. Franceschinis. Modeling with Generalized Stochastic Petri Nets. John Wiley & Sons Ltd., 1995.

[Marti & Garcia-Molina 2003]
S. Marti, H. Garcia-Molina. Identity Crisis: Anonymity vs. Reputation in P2P Systems. In *IEEE Conf. on Peer-to-Peer Computing*, pp. 134–141, IEEE CS Press, September 2003.

[Martin-Guillerez 2006]
D. Martin-Guillerez. Increasing Data Resilience of Mobile Devices with a Collaborative Backup Service. In *Supplemental Proc. of the Int. Conf. on Dependable Systems and Networks (DSN'06)*, pp. 139–141, 2006.

[Martinian 2004]
E. Martinian. Distributed Internet Backup System. MIT, USA, 2004. *http://dibs.sourceforge.net/*.

[Mavrogiannopoulos 2007]
N. Mavrogiannopoulos. Using OpenPGP Keys for Transport Layer Security Authentication (RFC 5081). Internet Engineering Task Force (IETF), November 2007. *http://tools.ietf.org/html/rfc5081*.

[Melski 1999]
E. Melski. Burt: The Backup and Recovery Tool. In *Proc. of the USENIX Conf. on System Administration*, pp. 207–218, USENIX Association, 1999.

[Menascé 2003]
D. A. Menascé. Security Performance. In *IEEE Internet Computing*, 7, June 2003, pp. 84–87.

[Mens 2002]
T. Mens. A State-of-the-Art Survey on Software Merging. In *IEEE Transactions on Software Engineering*, 28(5) , May 2002, pp. 449–462.

[Merkle 1980]
R. C. Merkle. Protocols for Public Key Cryptosystems. In *Proc. of the IEEE Symp. on Security and Privacy*, pp. 122–134, April 1980.

[Michiardi & Molva 2002]
P. Michiardi, R. Molva. CORE: A Collaborative Reputation Mechanism to Enforce Node Cooperation in Mobile Ad Hoc Networks. In *Proc. of the 6th IFIP TC6/TC11 Joint Conf. on Communications and Multimedia Security*, pp. 107–121, Kluwer Academic Publishers, September 2002.

[Milgram 1967]
S. Milgram. The Small World Problem. In *Psychology Today*, 2, 1967, pp. 60–67.

[Miller 2006]
M. S. Miller. Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control, PhD Thesis, Johns Hopkins University, Baltimore, MA, USA, May 2006.

[Mitzenmacher 2004]
M. Mitzenmacher. Digital Fountains: A Survey and Look Forward. In *Proc. of the IEEE Information Theory Workshop*, pp. 271–276, October 2004.

[Montenegro & Castelluccia 2002]
G. Montenegro, C. Castelluccia. Statistically Unique and Cryptographically Verifiable (SUCV) Identifiers and Addresses. In *Proc. of the Network and Distributed System Security Symp. (NDSS)*, 2002.

[Morcos *et al.* 2006]
F. Morcos, T. Chantem, P. Little, T. Gasiba, D. Thain. iDIBS: An Improved Distributed Backup System. In *Proc. of the Int. Conf. on Parallel and Distributed Systems (ICPADS)*, pp. 58–67, IEEE CS Press, July 2006.

[Mummert *et al.* 1995]
L. B. Mummert, M. R. Ebling, M. Satyanarayanan. Exploiting Weak Connectivity for Mobile File Access. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pp. 143–155, 1995.

[Muthitacharoen *et al.* 2001]
A. Muthitacharoen, B. Chen, D. Mazières. A Low-Bandwidth Network File System. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, pp. 174–187, October 2001.

[NESSIE Consortium 2003a]
NESSIE Consortium. NESSIE Security Report. Technical Report NES/DOC/ENS/WP5/D20/2, February 2003.

[NESSIE Consortium 2003b]
NESSIE Consortium. Portfolio of Recommended Cryptographic Primitives. Technical Report , February 2003.

[Nethercote & Seward 2007]
N. Nethercote, J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pp. 89–100, 2007.

[Nightingale & Flinn 2004]
E. B. Nightingale, J. Flinn. Energy-Efficiency and Storage Flexibility in the Blue File System. In *Proc. of the 6th Symp. on Operating Systems Design and Implementation (OSDI'04)*, December 2004.

[Oberhumer 2005]
M. F.X.J. Oberhumer. LZO Real-Time Data Compression Library. 2005. *http://www.oberhumer.com/opensource/lzo/*.

[One Laptop Per Child Project 2007]
One Laptop Per Child Project. Mesh Networking in OLPC. 2007. *http://wiki.laptop.org/go/Mesh_Network_Details*.

[Open Mobile Alliance 2001]
Open Mobile Alliance. SyncML Sync Protocol, Version 1.0.1. June 2001. *http://www.openmobilealliance.org/tech/affiliates/syncml/syncmlindex.html*.

[Oualha *et al.* 2007a]
N. Oualha, P. Michiardi, Y. Roudier. A Game Theoretic Model of a Protocol for Data Possession Verification. In *Proc. of the IEEE Int. Workshop on Trust, Security, and Privacy for Ubiquitous Computing (TSPUC)*, June 2007.

[Oualha *et al.* 2007b]
N. Oualha, S. M. Önen, Y. Roudier. Verifying Self-Organized Storage with Bilinear Pairings. Technical Report Research Report RR-07-201, Eurecom, France, June 2007.

[Papadopouli & Schulzrinne 2000]
M. Papadopouli, H. Schulzrinne. Seven Degrees of Separation in Mobile Ad Hoc Networks. In *IEEE Conf. on Global Communications (GLOBECOM)*, November 2000.

[Park *et al.* 2004]
C. Park, J-U. Kang, S-Y. Park, J-S. Kim. Energy-Aware Demand Paging on NAND Flash-Based Embedded Storages. In *Proc. of the Int. Symp. on Low Power Electronics and Design*, pp. 338–343, ACM Press, 2004.

[Peacock 2006]
A. Peacock. flud Backup: A Free Peer-to-Peer Backup System. 2006. *http://www.flud.org/*.

[Perkins *et al.* 2003]
C. E. Perkins, E. M. Belding-Royer, S. R. Das. RFC3561 – Ad hoc On-Demand Distance Vector (AODV) Routing. Internet Engineering Task Force, July 2003. *http://www.ietf.org/rfc/rfc3561.txt*.

[Peterson & Burns 2003]
Z.N.J. Peterson, R.C. Burns. Ext3cow: The Design, Implementation, and Analysis of Metadata for a Time-Shifting File System. Technical Report HSSL-2003-03, Hopkins Storage Systems Lab, Department of Computer Science, Johns Hopkins University, USA 2003.

[Plank *et al.* 1995]
J. S. Plank, M. Beck, G. Kingsley. Libckpt: Transparent Checkpointing Under Unix. In *Proc. of the USENIX Annual Technical Conf.*, USENIX Association, January 1995.

[Poettering 2006]
L. Poettering. The Avahi DNS-SD/mDNS Free Implementation. 2006. *http://avahi.org/*.

[Poettering 2007]
L. Poettering. Diensteverwaltung in Ad-Hoc-Netzwerken (Diplomarbeit). Fakultät für Mathematik, Informatik und Naturwissenschaften, Universität Hamburg, Germany, April 2007. *http://0pointer.de/public/diplom.pdf*.

[Preguica *et al.* 2005]
N. Preguica, C. Baquero, J. L. Martins, M. Shapiro, P. S. Almeida, H. Domingos, V. Fonte, S. Duarte. FEW: File Management for Portable Devices. In *Proc. of the Int. Workshop on Software Support for Portable Storage*, March 2005.

[Prevayler.Org 2006]
Prevayler.Org. Prevayler, a Java Implementation of "Object Prevalence". 2006. *http://www.prevayler.org/*.

[Quinlan & Dorward 2002]
S. Quinlan, S. Dorward. Venti: A New Approach to Archival Storage. In *Proc. of the 1st USENIX Conf. on File and Storage Technologies*, pp. 89–101, 2002.

[Rahman *et al.* 2006]
S. M. M. Rahman, A. Inomata, T. Okamoto, M. Mambo, E. Okamoto. Anonymous Secure Communication in Wireless Mobile Ad-hoc Networks. In *Proc. of the 1st Int. Conf. on Ubiquitous Convergence Technology*, pp. 131–140, Springer-Verlag, December 2006.

[Ranganathan *et al.* 2002]

K. Ranganathan, A. Iamnitchi, I. Foster. Improving Data Availability Through Dynamic Model-Driven Replication in Large Peer-to-Peer Communities. In *Proc. of the Workshop on Global and Peer-to-Peer Computing on Large Scale Distributed Systems*, pp. 376–381, IEEE CS Press, May 2002.

[Red Hat, Inc. 2007]

Red Hat, Inc.. Hibernate: Relational Persistence for Java and .NET. 2007. *http://www.hibernate.org/.*

[Rubel 2005]

M. Rubel. Rsnapshot: A Remote Filesystem Snapshot Utility Based on Rsync. 2005. *http://rsnapshot.org/.*

[Sailhan & Issarny 2005]

F. Sailhan, V. Issarny. Scalable Service Discovery for MANET. In *Proc. of the IEEE Int. Conf. on Pervasive Computing and Communication*, March 2005.

[Saltzer 1978]

J. H. Saltzer. Naming and Binding of Objects. In *Operating Systems, An Advanced Course*, pp. 99–208, Springer-Verlag, 1978.

[Santry *et al.* 1999]

D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, J. Ofir. Deciding When to Forget in the Elephant File System. In *Proc. of the 17th ACM Symp. on Operating Systems Principles*, pp. 110–123, December 1999.

[Scott & Burleigh 2007]

K. L. Scott, S. Burleigh. Bundle Protocol Specification (Internet Draft). The MITRE Corporation, VA, USA, April 2007. *http://tools.ietf.org/html/draft-irtf-dtnrg-bundle-spec-09.*

[Seward 2007]

J. Seward. Bzip2 and Libbzip2 Website. 2007. *http://www.bzip.org/.*

[Shapiro & Adams 2002]

J. S. Shapiro, J. Adams. Design Evolution of the EROS Single-Level Store. In *Proc. of the USENIX Annual Technical Conf.*, May 2002.

[Shapiro & Vanderburgh 2002]

J. S. Shapiro, J. Vanderburgh. CPCMS: A Configuration Management System Based on Cryptographic Names. In *Proc. of the USENIX Annual Technical Conf., FREENIX Track*, pp. 207–220, USENIX Association, 2002.

[Shivers 1999]

O. Shivers. SRFI-1 — List Library. Massachusetts Institute of Technology, Cambridge, MA, USA, October 1999. *http://srfi.schemers.org/srfi-1/srfi-1.html.*

[Sit *et al.* 2003]

E. Sit, J. Cates, R. Cox. A DHT-based Backup System. Technical Report , MIT Laboratory for Computer Science, August 2003.

[Soules *et al.* 2003]
C. A.N. Soules, G. R. Goodson, J. D. Strunk, G. R. Ganger. Metadata Efficiency in Versioning File Systems. In *Proc. of the 2nd USENIX Conf. on File and Storage Technologies*, April 2003.

[Spyropoulos *et al.* 2007]
T. Spyropoulos, K. Psounis, C. Raghavendra. Efficient Routing in Intermittently Connected Mobile Networks: The Single-Copy Case. In *ACM/IEEE Transactions on Networking*, , 2007, .

[Srinivasan 1995]
R. Srinivasan. RPC: Remote Procedure Call Protocol Specification, Version 2 (RFC 1831). Internet Engineering Task Force (IETF), August 1995. *http://tools.ietf.org/html/rfc1831*.

[Stefansson & Thodis 2006]
B. Stefansson, A. Thodis. MyriadStore: A Peer-to-Peer Backup System. Technical Report Master of Science Thesis, Stockholm, Sweden, June 2006.

[Stemm *et al.* 1997]
M. Stemm, P. Gauthier, D. Harada, R. H. Katz. Reducing Power Consumption of Network Interfaces in Hand-Held Devices. In *IEEE Transactions on Communications*, E80-B(8) , August 1997, pp. 1125–1131.

[Sözer *et al.* 2004]
H. Sözer, M. Tekkalmaz, I. Körpeoglu. A Peer-to-Peer File Sharing System for Wireless Ad-Hoc Networks. In *Proc. of the 3rd Annual Mediterranean Ad Hoc Networking Workshop (MedHoc)*, June 2004.

[Tanenbaum *et al.* 1986]
A. S. Tanenbaum, S. J. Mullender, R. v. Renesse. Using Sparse Capabilities in a Distributed Operating System. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, pp. 558–563, May 1986.

[The UbiSec Project 2005]
The UbiSec Project. Ubiquitous Networks with a Secure Provision of Services, Access, and Content Delivery. 2005. *http://jerry.c-lab.de/ubisec/*.

[Tichy 1985]
W. F. Tichy. RCS—A System for Version Control. In *Software—Practice & Experience*, 15, New York, NY, USA, 1985, pp. 637–654.

[Tolia *et al.* 2003]
N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, T. Bressoud, A. Perrig. Opportunistic Use of Content Addressable Storage for Distributed File Systems. In *Proc. of the USENIX Annual Technical Conf.*, pp. 127–140, 2003.

[Tolia *et al.* 2004]
N. Tolia, J. Harkes, M. Satyanarayanan. Integrating Portable and Distributed Storage. In *Proc. of the USENIX Conf. on File and Storage Technologies*, pp. 227–238, March 2004.

[Tridgell & Mackerras 1996]
A. Tridgell, P. Mackerras. The Rsync Algorithm. Technical Report TR-CS-96-05, Department of Computer Science, Australian National University Canberra, Australia 1996.

[Tridgell *et al.* 1999]
A. Tridgell, P. Russel, J. Allison. The Trivial Database. 1999. *http://samba.org/*.

[Vernois & Utard 2004]
A. Vernois, G. Utard. Data Durability in Peer to Peer Storage Systems. In *Proc. of the 4th Workshop on Global and Peer to Peer Computing*, pp. 90–97, IEEE CS Press, April 2004.

[Wang *et al.* 2005a]
Y. Wang, S. Jain, M. Martonosi, K. Fall. Erasure-Coding Based Routing for Opportunistic Networks. In *Proc. of the ACM SIGCOMM Workshop on Delay-Tolerant Networking*, pp. 229–236, ACM Press, 2005.

[Wang *et al.* 2005b]
X. Wang, Y. Yin, H. Yu. Finding Collisions in the Full SHA-1. In *Proc. of the CRYPTO Conf.*, pp. 17–36, 2005.

[Wang *et al.* 2006]
Y-H. Wang, C-F. Chao, S-W. Lin, W-T. Chen. A Distributed Data Caching Framework for Mobile Ad Hoc Networks. In *Proc. of the 2006 Int. Conf. on Communications and Mobile Computing*, pp. 1357–1362, ACM Press, 2006.

[Weatherspoon & Kubiatowicz 2002]
H. Weatherspoon, J. Kubiatowicz. Erasure Coding vs. Replication: A Quantitative Comparison. In *Revised Papers from the 1st Int. Workshop on Peer-to-Peer Systems*, pp. 328–338, Springer-Verlag, 2002.

[Widmer & Boudec 2005]
J. Widmer, J-Y. L. Boudec. Network Coding for Efficient Communication in Extreme Networks. In *Proc. of the ACM SIGCOMM Workshop on Delay-Tolerant Networking*, pp. 284–291, ACM Press, 2005.

[Xu *et al.* 1999]
L. Xu, V. Bohossian, J. Bruck, D. G. Wagner. Low Density MDS Codes and Factors of Complete Graphs. In *IEEE Transactions on Information Theory*, 45(1) , November 1999, pp. 1817–1826.

[Xu 2005]
L. Xu. Hydra: A Platform for Survivable and Secure Data Storage Systems. In *Proc. of the ACM Workshop on Storage Security and Survivability*, pp. 108–114, ACM Press, November 2005.

[You & Karamanolis 2004]
L. L. You, C. Karamanolis. Evaluation of Efficient Archival Storage Techniques. In *Proc. of 21st IEEE/12th NASA Goddard Conf. on Mass Storage Systems and Technologies*, pp. 227–232, April 2004.

[You *et al.* 2005]
L. L. You, K. T. Pollack, D. D. E. Long. Deep Store: An Archival Storage System Architecture. In *Proc. of the 21st Int. Conf. on Data Engineering (ICDE 2005)*, pp. 804–815, IEEE CS Press, April 2005.

[Zhang *et al.* 2005]
T. Zhang, S. Madhani, P. Gurung, E. v. d. Berg. Reducing Energy Consumption on Mobile Devices with WiFi Interfaces. In *Proc. of the IEEE Global Telecommunications Conf. (Globecom)*, pp. 561–565, IEEE CS Press, December 2005.

[Zhang 2006]
Z. Zhang. Routing in Intermittently Connected Mobile Ad Hoc Networks and Delay Tolerant Networks: Overview and Challenges. In *IEEE Communications Surveys & Tutorials*, 8, January 2006, pp. 24–37.

[Zheng & Lee 2004]
J. Zheng, M. J. Lee. Will IEEE 802.15.4 Make Ubiquitous Networking a Reality? A Discussion on a Potential Low Power, Low Bit Rate Standard. In *IEEE Communications Magazine*, 42, June 2004, pp. 140–146.

[Zöls *et al.* 2005]
S. Zöls, R. Schollmeier, W. Kellerer, A. Tarlano. The Hybrid Chord Protocol: A Peer-to-Peer Lookup Service for Context-Aware Mobile Applications. In *Proc. of the Int. Conf. on Networking*, Lecture Notes in Computer Science, pp. 781–792, Springer Berlin/Heidelberg, 2005.

Colophon

This document was authored using Skribilo and typeset using Lout. The typeface of the body of the text is Gentium, by Victor Gaultney and SIL International. Section headings were typeset using the Charis typeface by SIL International, while chapter headings were typeset using the Latin Modern Sans Sérif typeface designed by Donald E. Knuth.

Sauvegarde coopérative de données pour dispositifs mobiles

Cooperative Data Backup for Mobile Devices

Thèse de Doctorat

Ludovic Courtès

Résumé

Les dispositifs informatiques mobiles tels que les ordinateurs portables, assistants personnels et téléphones portables sont de plus en plus utilisés. Cependant, bien qu'ils soient utilisés dans des contextes où ils sont sujets à des endommagements, à la perte, voire au vol, peu de mécanismes permettent d'éviter la perte des données qui y sont stockées. Dans cette thèse, nous proposons un service de sauvegarde de données coopératif pour répondre à ce problème. Cette approche tire parti de communications spontanées entre de tels dispositifs, chaque dispositif stockant une partie des données des dispositifs rencontrés. Une étude analytique des gains de cette approche en termes de sûreté de fonctionnement est proposée. Nous étudions également des mécanismes de stockage réparti adaptés. Les problèmes de coopération entre individus mutuellement suspicieux sont également abordés. Enfin, nous décrivons notre mise en oeuvre du service de sauvegarde coopérative.

Mots-clef : sûreté de fonctionnement, informatique ubiquiste, sauvegarde de données, systèmes pair-à-pair

Summary

Mobile devices such as laptops, PDAs and cell phones are increasingly relied on but are used in contexts that put them at risk of physical damage, loss or theft. However, few mechanisms are available to reduce the risk of losing the data stored on these devices. In this dissertation, we try to address this concern by designing a cooperative backup service for mobile devices. The service leverages encounters and spontaneous interactions among participating devices, such that each device stores data on behalf of other devices. We first provide an analytical evaluation of the dependability gains of the proposed service. Distributed storage mechanisms are explored and evaluated. Security concerns arising from the cooperation among mutually suspicious principals are identified, and core mechanisms are proposed to allow them to be addressed. Finally, we present our prototype implementation of the cooperative backup service.

Keywords: dependability, ubiquitous computing, data backup, peer-to-peer systems