Data synchronization and browsing for home environments

DOI:
10.6100/IR576805

Document status and date:
Published: 01/01/2004

Data Synchronization and Browsing for Home Environments

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Qian, Yuechen

Data synchronization and browsing for home environments /
by Yuechen Qian. -
Eindhoven : Technische Universiteit Eindhoven, 2004.
Proefschrift. - ISBN 90-386-0832-2
NUR 986
Subject headings : distributed systems / logging and recovery / formal methods / file organization / image databases / information browsers
CR Subject Classification (1998) : C.2.4, H.2.7, D.2.4, H.3.2, H.2.8, H.4.3

Printed by University Press Facilities, Eindhoven, The Netherlands
Cover design by Yuechen Qian and Hennie Alblas

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Dedicated to Ting,
Shijun and Wenqi, and the memory of Yuexun.

Data Synchronization and Browsing for Home Environments

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr. R.A. van Santen, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op woensdag 2 juni 2004 om 16:00 uur

door

Yuechen Qian

geboren te Wuxi, China

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.ir. L.M.G. Feijs
en
prof.dr. J.C.M. Baeten

Copromotor:
dr.ir. M.P. Bodlaender

Preface

Thanks to the rapid advance of network technologies and device miniaturization, interconnected consumer electronic devices are appearing in home environments and are becoming ubiquitous. A device or a system that is truly appreciated by consumers does not come about without much effort. It requires both seamless integration of the state-of-the-art technologies and careful consideration of user requirements in home environments. As a research attempt of this sort, the "Perceptive Home Environments" (Phenom) project was launched in 1999 by Philips Research (the Netherlands) and Eindhoven University of Technology, to investigate technological solutions for helping people in their social activities in home environments. I was happy with the opportunity to join the Phenom project as a Ph.D. student, working on the design and prototyping of a distributed data management system for home environments.

In the past four years I have worked closely with the other members of the Phenom project at Philips Research Laboratories Eindhoven. Together, we developed several new concepts for digital photo browsing in home environments. Those concepts were demonstrated at Philips Corporate Research Exhibitions and during several European exhibition events. I published several papers on various aspects of the distributed system that I developed in the Phenom project, together with my supervisors and advisors. The book in your hands is a reflection of my four years' work.

I thank all those who helped me in one way or another in my Ph.D. study. In particular I owe great thanks to prof.dr.ir. Loe Feijs, who gave me the opportunity to participate in the Phenom project and supervised my study in the past years. His broad research interests and proficiency in many areas inspired me in many aspects of my research, his rigorous research attitude showed me what a scientist's true spirit should be like, and his kindness always encouraged me to move forward. I would also like to thank prof.dr. Jos Baeten. His strictness and perfectionism helped me add the finishing touch to this book.

I would like to thank dr.ir. Maarten Bodlaender and dr.ir. Rob Udink of Philips Research Laboratories Eindhoven. As the industrial advisors of my research, they were always available to help me with their professional knowledge and expertise.

My research was carried out at the Media Interaction Group of Philips Research. I thank my colleagues there for my pleasant stay. In particular I would like to thank the Phenom team: Evert van Loenen, Esko Dijk, Nick de Jong, Elise van den Hoven, Dario Teixeira, Yvonne Burgers and Doug Tedd. Working in this multidisciplinary team not only broadened my knowledge, but also taught me how to approach my work from different angles. My thanks also go to Ramon Clout for his advice on preparing this thesis.

My work would not have been done without support from the staffs of Eindhoven Embedded Systems Institute and the Faculty of Industrial Design. I owe special thanks to dr.ir. Marloes van Lierop for her generous support during my study and Helen Maas for her emotional support and kind help in the preparations.

Finally my thanks go to my family. I would like to thank my wife, Ting. Her persistent encouragement and tolerance is the source of my confidence. Thanks to all the time and effort she spent at our home, I was able to concentrate on my work. I would like to thank my parents, Shijun and Wenqi. My father allowed me to make my first keystroke on his office computer when I was thirteen. And later he paved the way towards computer science for me. My mother's gentle criticism always reminds me of the importance of self-reflection.

Yuechen Qian

Eindhoven, The Netherlands
December 2003

Contents

Part I

Background and problem analysis

1. Introduction

At the time of writing this thesis, the concept of distributed data management is not well understood by end users in home environments. Nevertheless, people are gradually getting used to file sharing, video conferencing, Internet gaming, distance learning and other applications thanks to the rapid progress made in network technologies and device miniaturization. Many concepts that were invented during the evolution of distributed systems will appear in home environments. This thesis hopes to contribute towards that development.

As network technologies advance, it is becoming technically feasible to connect all personal devices to home networks and the Internet. In future home environments, people will be able to access personal information on any device, anywhere and anytime. People will not only access data at home, but also on the move. This vision is described by *ambient intelligence* [1] and *ubiquitous computing* [172]. In an attempt to help realize this vision, this thesis presents the design of a distributed data management system that will help people manage personal data in home environments.

1.1 Background

Under the umbrella of ambient intelligence, Eindhoven University of Technology and Philips Research Laboratories Eindhoven launched the "Perceptive Home Environments" (Phenom) project [77, 166]. Phenom focused on user-system interaction in future home environments and investigated how intelligent home environments can help people to easily browse, find and share personal memories.

Phenom was implemented in two phases. In the first phase, four research areas were defined. They were user-system interaction, system adaptivity, software architecture and device architecture. To obtain a better understanding of these research areas, the "Memory Browser" system [33] was designed and implemented as an experiment.

The "Memory Browser" system is an intelligent photo browsing system that can recognize multiple users, devices and objects, and learn from their behaviors. Users of this system can not only browse photos on a touch screen (Sepia), but also show photos on displaying devices in the vicinity. Moreover, the system enables users to easily retrieve photos by using souvenirs.[1] In the system, a user can associate a collection of digital photos with a souvenir object and can retrieve them simply by putting the object on a table. The table recognizes the object and notifies the system to show photos associated with the object.

Based on the evaluation of the "Memory Browser" system, the research topics of Phenom were refined. In the second phase of the project, Phenom focused on the following five research directions: human-memory recollection, conversational search, tracking and localization, data management and middleware platforms. As an integration of the research in the five directions, an intelligent "Memory Sharing" system [34] was developed.

The "Memory Sharing" system was designed to address people's need of sharing personal memories. By exploiting on-device storage, Sepias enable users to take collections of photos with them. In this way, users can enjoy photos on the move and share them with friends. When Sepias are brought back home, modifications to photos are synchronized so that the users have consistent views on their personal photos.

This thesis focuses on data management, especially the design of a distributed data system for home environments. The other four aspects defined in the second phase of Phenom, namely human-memory recollection, conversational search, tracking and localization and middleware platforms, can be found in [164, 154, 32, 30], respectively.

1.2 Data management in home environments

The amount of information that people can access is increasing dramatically. People have all kinds of data in home environments. Typical examples of data are the following:

- Digital assets, such as digital photos, music, movies, MP3 files, videos.
- Personal documents, such as agendas, e-mails, shopping lists, things-to-do lists.
- Programs, such as executables and libraries of the applications for word processing, image editing, photo browsing, music playback, and gaming.
- User profiles, such as files that record user preferences, graphical user interface (GUI) customization, system adaptation and statistics.

Data are often stored in different devices, which are interconnected via wired and/or wireless home networks. Some devices are stationary while others can be carried around. Some devices always reside at home while others

[1] Such object-based information access [59] is user-friendly in home environments. This concept was also exploited in the Philips' LiMe project[76]. Users of the LiMe system can share and exchange information in communities by using coin-sized tags.

can be brought in and out on a daily basis. Data can also be stored on the Internet. A variety of storage services will converge in home environments to serve the user's needs. There is a rapidly growing need for systems that help people to easily manage their data in home environments.

Distributed data management systems have a long history in computer science and industries. Home environments have characteristics that are different from those of professional environments. Seven challenges [41] have been identified: accidental smartness, impromptu interoperability, no systems administrator, designing for domestic use, social implications of aware home technologies, reliability, and inference in the presence of ambiguity.

More specific to data management, the following aspects of home environments are identified in the research context of this thesis. They introduce additional requirements and opportunities in designing a distributed data management system for home environments.

Heterogeneity. Data sources, such as file systems, database systems and Internet services, are heterogeneous in their nature. Moreover, devices that provide data may run different operating systems, have different resource constraints, come from different vendors, and require different administrative efforts. This is related to the challenge of impromptu interoperability.

Dynamics. Devices are switched on and off frequently. New devices are brought in and old ones are moved out. Such device activities can be performed anytime, at the user's will. This is related to the challenges of reliability, impromptu interoperability and no system administrators.

Ease-of-use. In-home data management systems should not require the user to have technical knowledge. The systems should not involve time-consuming efforts during installation, configuration and administration. This is related to the challenge of no system administrator.

To tackle heterogeneity and dynamics of real environments, not only for home environments, several middleware platforms have been built. UPnP [158] provides mechanisms for network services discovery in local area networks. OSGi [157] is an open specification for network delivery of Internet services to local networks and devices in home environments. Jini [145], which is based on Java [144], provides a homogeneous platform for achieving interoperability between services in mobile environments. HAVi [156] is based on the IEEE 1394 protocol and is designed for real-time streaming applications in local area networks. DCOM [95] allows software components to cooperate seamlessly in local and wide area networks.

No data management services are specified in those platforms. System engineers have to build functional components that take care of storage services in middleware platforms. For example, computing devices with storage, either file systems or database systems, export their storage space in the form of the so-called *content directories* in UPnP. Storage services on the Internet are available through proxies in those platforms.

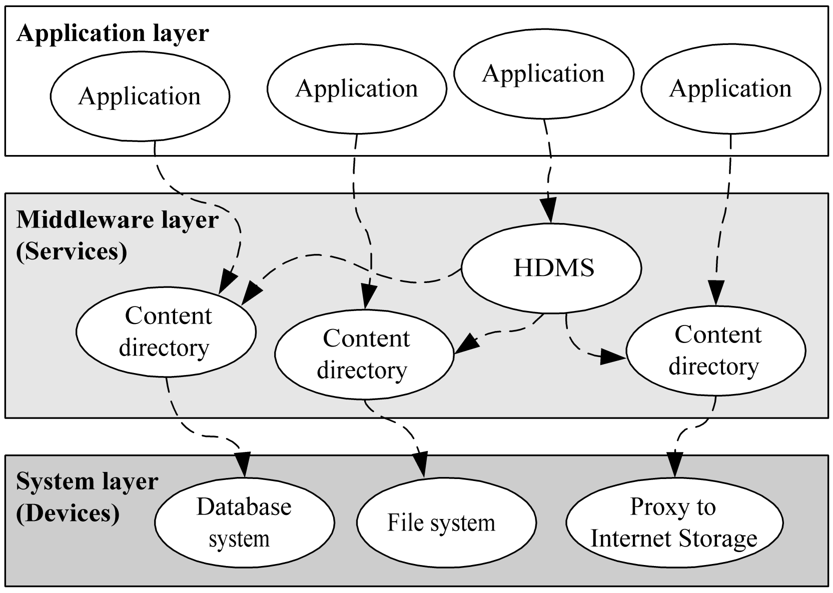


Figure 1.1. An overview of an in-home distributed data management system.

An in-home data management system (HDMS) is responsible for storing programs and data and making them available as needed. In precise terms, an HDMS is responsible for data management of content directories in a home network. The term "management" here stresses the fact that the data objects stored in different devices are managed in a coordinated way in a system, not only in a stand-alone manner in individual devices.

Figure 1.1 gives an architectural overview of an HDMS. In the figure, storage devices and services in a home environment reside in the system layer. They appear as content directories in a middleware platform. An HDMS is a functional component in the middleware layer, responsible for managing content directories. Applications in the application layer selectively access data via an individual content directory or the HDMS.

An HDMS hides device- or operating-system-specific information from application developers and provides a unified interface for accessing contents. In this way, software engineers can concentrate on functionality design of a home system. Moreover, with an HDMS, devices can be coordinated to show functionalities and features that individual devices could never achieve on their own, such as data sharing, data migration, data replication among devices.

1.3 Eventual consistency

An important issue in distributed data management systems is data replication. In traditional distributed systems, data are often replicated to enhance reliability or improve performance. In home environments, data are largely replicated to improve availability. Devices can sometimes be disconnected from home networks and/or the Internet. This may be done for a variety of reasons such as costs, security, privacy, human factors, technical restrictions, partitioning, power shutdown or technical failures.

Moreover, thanks to rapid developments in storage technologies and device miniaturization, people can store a large amount of information on portable devices, such as laptops, PDAs, MP3 players and portable hard disks. Portable devices can be connected to other devices, home networks and/or the Internet anytime at the user's own will. This is the so-called *intermittent connectivity* [116].

When devices are disconnected from home networks for whatever reasons, people can continue to access the data stored on the devices. They can also modify the data, if necessary. Data can be updated independently of their copies.

Definition 1.3.1 (Disconnected updates). *Disconnected updates are data modifications that are made while devices are disconnected.*[2]

Without precautions, data inconsistencies may be introduced. For example, one photo on a PDA was deleted while a copy of it on a home server was cropped. Both operations were performed when the PDA and the home server were disconnected. Such conflicting updates introduce inconsistent views on the photo. Inconsistencies should be eliminated to avoid confusion.

Definition 1.3.2 (Data synchronization). *Data synchronization is the process of resolving inconsistencies of disconnected updates.*[3]

[2] Disconnected updates can be regarded as the data modifications that are made in a disconnected operation. The term "disconnected operation" was originally introduced in the Coda system [74]. It stands for a mode of a distributed file system, in which clients are allowed to access locally cached data during device disconnection.

[3] The term "data synchronization" was borrowed from the name of file synchronization tools such as the Unison File Synchronizer [112]. While those tools are designed for file management, data synchronization focuses on how to resolve inconsistencies of data objects in general. This term is close to "data reconciliation" in replicated file systems. Moreover, the term "data synchronization" should not be confused with operation synchronization in distributed shared memory systems. In those systems, synchronization has the meaning of scheduling data access operations performed by different processors. Different from operation synchronization, data synchronization deals with propagation of data updates between devices.

To resolve data inconsistencies, for instance, logs can be used to record disconnected updates. The information captured in logs is used in data synchronization.

Keeping replicated data consistent is one of the key problems for an HDMS. *Consistency* means that different copies of a file should have the same content. A survey of various consistency models can be found in [150, 152]. To what extent consistency needs to be guaranteed in a system may vary from application to application. There is always a tradeoff between availability and consistency in system design.

In recently developed very large distributed systems, such as peer-to-peer systems, users show tolerance to a certain degree of inconsistency to benefit from high data availability. Those systems have one form of consistency in common, *eventual consistency* [133, 152].

Definition 1.3.3 (Eventual consistency). *If no updates take place and the devices containing data copies can communicate freely for a long time, all data copies will eventually become consistent.*

Eventual consistency places little requirement on a replication algorithm. Without this guarantee, the replicated contents may remain inconsistent for ever, making a system practically useless. An eventually consistent system usually makes a best effort to disseminate updates among copies. Such a “best effort” is strong and practical enough for many applications.

Eventual consistency is a suitable consistency model for an HDMS. In the first place, maintaining data consistency is a user-centric task in home environments. People want to keep their data under their own control. Data synchronization that is based on eventual consistency is often implemented in a peer-to-peer manner, which can be easily understood by the user. The user can perform data synchronization anytime at his/her own will. Secondly, home environments have a high degree of heterogeneity of resource constraints. Maintaining eventual consistency requires minimum system resources and network bandwidth. Thirdly, there are frequent device disconnections and device reconnections in home environments. In such dynamic and loosely connected systems, traditional consistency models are hard to implement. Fourthly, many applications in home environments do not require tightly-coupled operations, such as transactions that are used in banking systems and flight reservation systems. Quantitative evidence [74] shows that the average level of write-sharing in personal computing environments is low. And finally, eventual consistency is in general simple to implement and does not require much system administrative effort, which coincides with the easy-of-use requirement of home environments.

1.4 Research objective

The objective of this thesis is the design of a distributed data management system that achieves eventual consistency for home environments. Due to the research context of this thesis, the system is intended to support digital photo management and photo-related applications.

Designing a distributed data management system involves careful consideration in many aspects. For example, how to address the heterogeneity of home environments? How to deal with device in/out and device on/off activities in home environments? How to select the data to be replicated onto portable devices that people carry with them? How to propagate updates on data copies when devices are connected? How should systems be installed and maintained without much effort? Several of these aspects or combinations of them have been addressed in various systems (see Section 1.6). The main strategy here is to find a proper solution among them, possibly tailored, for home environments.

With respect to the eventual consistency requirement referred to in Section 1.3, the following aspects need to be addressed.

- Log management: How to record disconnected updates on data copies when devices are disconnected?
- Data synchronization: What are data inconsistencies? How to detect and resolve data inconsistencies when devices are reconnected?

So far, log management and data synchronization have been studied empirically. This thesis aims to place a formal foundation under existing works on data synchronization.

1.5 Formal methods

Formal methods are mathematically based techniques for describing and reasoning about system properties. They provide frameworks within which people can specify, develop and verify systems in a systematic and rigorous manner. A formal method has a sound mathematical basis, given by a formal specification language. Such a language provides a means of precisely defining notations, such as consistency, completeness and correctness, and rigorously verifying those properties.

There is a wide diversity of formal methods for system, software and hardware development. To name a few, communicating sequential processes (CSP) [58], process algebra [8] and π-calculus [98] are theories on concurrent communication processes. Spin [60] is a model-checking system for studying communication systems. Module algebra [13] and component algebra [44] are algebraic calculi for studying compositional properties of software modules and components in software engineering. ASF+SDF is a development environment for component-based language development [163]. In practice, Z [140, 141], VDM [7] and COLD [42, 43] are model-based specification languages, widely used in system specification and design. PVS [134] and Isabelle [105] are mechanized verification tools supporting model-based specification, proof and verification. Proof theory [55] is used in industries for verifying that a code fragment can be executed safely on a host system [104]. The Java Modelling Language (JML) [89] is used to include annotations (correctness assertions) in Java classes and interfaces (as special comments), which is used in formalizing the JavaCard API in [162].

In this thesis, the Z [140, 141] specification language is used. Z is a model-based specification language. It is based on first-order logic and provides a rich set of mathematical machinery for system specification and design. Z was chosen as the working language on the basis of the following considerations. In the first place, consistency is data-centric. Data synchronization often requires semantic knowledge of how data are used. Little formal research has been carried out in this area. Generally speaking, resolving inconsistencies can be regarded as a transition between system states, for which the expressive power of Z is sufficient. Secondly, model-based formal methods are constructive in the sense that they help to establish algorithmic solutions in a formal way. Due to this constructiveness, models that are developed in this way can be used as starting points in detailed system design and implementation. Moreover, the models themselves can often be regarded as a reference implementation in system testing and verification.

1.6 Related work

Data replication improves performance and availability. In practice, realizing the benefits of data replication is difficult since data correctness must be maintained. One type of replicated systems offers *one-copy* semantics.[4] A user always has a single, highly available copy of a data object. Systems are responsible for keeping the replicas identical all the time. Algorithms providing the one-copy semantics usually prohibit access to a replica unless the contents of the replica are provably up-to-date. Those algorithms are called *pessimistic* in [133].

Optimistic replication [17, 85] achieves high availability by allowing operations whenever a single data replica is available, the so-called *one-copy availability* semantics [51]. Optimistic replication makes systems more tolerant to disconnection, network partitioning, or failure. This method has been widely used in distributed systems supporting mobile users.

The survey of related work covers both pessimistic and optimistic replication algorithms and systems. It focuses mainly on optimistic replication, since

[4] The analogous database term is *one-copy serializability* [19]. In database systems, one-copy serializability of an execution of several concurrent transactions means the execution produces the same database state as some serial execution of the same transactions.

optimistic replication uses eventual consistency as its consistency model. For each optimistically replicated system, update propagation, log management for disconnected updates, and data synchronization algorithms will be discussed, if applicable. File synchronization tools and peer-to-peer file-sharing systems relate to this thesis and will therefore also be discussed.

1.6.1 Pessimistic replication

Algorithms like *primary copy replication*, *voting* [48], *available copies* [18], and *voting with ghosts* [167] are used to achieve the one-copy semantics of replicated data. For example, the primary copy replication algorithm relies on a primary server. The voting algorithm prohibits clients from updating data when a few servers are down. The available copies algorithm and the voting with ghosts algorithm do not handle network partitions and disconnections. A survey of solutions to the consistency problem of distributed database systems in partitioned networks can be found in [19, 23, 29].

The above-mentioned algorithms have been implemented in many systems. Harp [93] implements the primary copy algorithm for file replication. Harp is designed to improve the performance of file systems. In each replication server, Harp records modifications in a log residing in volatile memory. Operations in the log are applied to the file system in the background. Harp relies on a write-back strategy of update propagation. To avoid power failure, each server is equipped with an uninterrupted power supply. Harp runs only at the server side and is independent of any caching that may occur on the client side. Neither does Harp survive server shutdown or crash, nor does it allow mobile users to access files during disconnection.

Locus [171] is a distributed system supporting automatic management of replicated data, with the ability to withstand failures and network partition. In Locus, file replication serves mainly to increase availability for reading purposes. The primary copy approach is adopted for modifications. During network partitions, read requests are served as long as one copy of a file is available. It is guaranteed that the version read is the most recent one available in a partition. Updates are allowed only in a partition that has the primary copy. Locus withstands failures and network partition to a certain extent. It does not have mechanisms such as logging and conflict resolution to tackle network disconnection.

An asynchronous approach to replica control, *epsilon-serializability* [118], allows inconsistent data to be seen. In a system preserving one-copy serializability, read and write operations of transactions should be serialized. In a system preserving epsilon-serializability, read operations are allowed to be executed concurrently. This asynchronous approach is proposed under the assumption that federated databases may not wish to support tight coupling for performance concerns. However, it does not show how it can be applied in database systems with disconnected operations.

1.6.2 Optimistic replication

Ficus is built on top of existing file systems to accommodate data replication [51]. Ficus uses an optimistic replication approach, one-copy availability, in conjunction with automatic propagation and directory reconciliation [117]. A physical layer receiving an update makes an entry in a local *new version cache*. A propagation daemon consults this cache and propagates updates when necessary.

In Ficus, each logical file is represented by a set of physical replicas. Each replica bears a file identifier that globally uniquely identifies the logical file and a replica identifier that uniquely identifies that replica. Each replica has a *version vector* that encodes the update history of the replica.

During conflict resolution, Ficus handles update/update conflicts, name conflicts, and remove/update conflicts. Ficus allows the user to specify how to resolve update/update conflicts of files. Moreover, it allows users to resolve content conflicts of a file and its replicas, depending on the type of file. Ficus uses *no lost update* semantics to handle remove/update conflicts [126]. Regarding name conflicts, Ficus keeps a logical file name valid in every replication site of the file unless it has been removed from every replication site. In this way, Ficus solves the create/delete ambiguity [52]. In Ficus, creating a hard link to a file adds a second name for the file. Renaming a file is treated as a remove operation followed by a create operation.

Rumor [53] is a peer-to-peer, reconciliation-based optimistically replicated file system designed for use in mobile computers. Rumor uses a peer model that allows opportunistic update propagation among sites, which is a good choice when connectivity patterns of portable devices are less predicable. Rumor operates entirely at the application level, providing facilities for file replication. Rumor maintains data consistency purely by periodic reconciliation, as in lazy replication. It uses version vectors to detect updates and enforces no lost update semantics conflict resolution, as Ficus [117] did.

Coda is intended for document preparation and program development in mobile environments. Coda improves data availability via server replication and client caching [74, 103]. Coda allows pair-wise data reconciliation between server replicas and between clients and servers, but not between clients. Coda uses a call-back mechanism for validating caches. Coda uses write-through in connection mode and write-back in disconnection.

Coda provides a framework for invoking *application-specific resolvers* (ASRs) to resolve concurrent updates to a file in different network partitions [83]. Knowledge needed for file resolution is encapsulated in terms of rules. Coda defines a simple language for specifying rules. Regarding directory conflicts, Coda uses logging of directory updates [84]. Coda provides more directory operations than Ficus. In the Coda system, a directory can be created, renamed, and hard-linked. Name/name conflicts, remove/update conflicts, update/update conflicts, and rename/rename conflicts are handled in Coda. To resolve those conflicts, Coda uses *resolution logs.* A resolution log is associated with each volume of data at a replication device. In Coda files have unique identifiers. Remove/update conflicts are resolved by using Coda's *version vectors*. In resolving directory conflicts, Coda uses *latest common entry* (LCE) to determine the changes needed to be propagated. Coda uses *lazy resolution*, that is, conflict resolution is only invoked in the course of serving a system call, when the system discovers that a file has divergent replicas.

In Coda, log sizes are limited. Limits of log files are derived from empirical data. In the case of log overflow, the earliest entries in logs are replaced by the most recent operations. To compress logs, Coda uses LCE to discard log entries that have become useless in synchronization.

The Andrew File System (AFS) [61] has been modified to handle disconnection [65] and partial connection [66]. Cache management of AFS is modified so that operations performed during disconnection are logged. On reconnection, operations on logs are replayed on AFS file servers. In the case of conflicting updates, data on the client side are stored onto servers using new names, which are reported to users. No log optimization is applied. When network bandwidth is low, AFS uses background replay to propagate deferred operations in logs to servers.

AFS uses the peephole log optimization method to compress logs. In this method, semantic rules are defined to eliminate Log entries that record temporal file creation, file deletion, and repetitive file updates.

DOC [62] uses two-level client caching to deal with disconnection. In connection mode, write-on-close is used as the cache modification policy and verify-before-use as the cache validation policy.

DOC has a log at each DOC client, keeping track of the time and the data of a file before and after modification. On reconnection, write/write conflicts and read/write conflicts are reported to the users. The user must choose a proper action, either overwrite the information on the server, delete the information in local cache, or copy the updated file in local cache to a new unique name.

PFS [37] provides an interface for mobility-aware applications to direct the file system in its caching and consistency decisions to exploit intermittent connectivity. PFS provides normal I/O file access functions with two additional arguments: level of consistency and callback function. Consistency levels can be write-through, write-back, write-local, read-consistent or read-local. The callback function defines the behavior of applications in the case of failures.

PFS is filename-based and does not have logs to support disconnection. PFS does not resolve conflicts, such as write/write conflicts, and will only notify the PFS client of the conflicts.

Rover [70] is an object-oriented approach that addresses disconnection and intermittent connectivity. It provides relocatable dynamic objects (RDOs) and queued remote procedure call (QRPC) for mobile applications.

In Rover, an RDO is an object with a well-defined interface that can be loaded into a client computer. Object consistency is provided by application-level locking or by using application-specific conflict resolution.

Modifications on cached RDOs are tentatively committed at the local device, until fully committed at the server. QRPC permits an application to continue functioning during disconnection. In Rover, Universal Resource Names [139] are used for uniquely naming RDOs. Rover uses operation logs to record QRPC operations. To address the log overflow problem, applications can filter out duplicate requests and can overwrite the appending operation of logs.

Bayou [110] uses the *anti-entropy* protocol for update propagation between weakly connected replicas.

Bayou uses *write-log* to record disconnected updates and *accept-timestamps* to totally order operations in write logs. On reconnection, Bayou propagates write logs, instead of database contents, resulting in a reduction in bandwidth for transporting content. It also helps to resolve the ambiguity introduced by creation and deletion. In Bayou, write logs can be truncated whenever desired. To detect whether log truncation causes loss of uncommitted writes, version vectors are installed at each server. If there is any loss of uncommitted writes, full database transfer has to occur. In this case, rolling back operations are needed.

The Bayou system is a platform of replicated database systems supporting variable-consistency of mobile applications [31]. Bayou supports application-specific mechanisms to detect and resolve the update conflicts, ensuring that replicas move towards eventual consistency. It includes dependency checks and per-write conflict resolution based on client-provided merge procedures. Bayou servers are able to roll back the effects of previously executed writes and redo them according to a global serialization order [153].

TACT [176] is a toolkit for distributed database systems, providing dynamic tuning of availability and consistency. It is designed for wide area networks. Write logs are used for replicas on servers and anti-entropy sessions are used to reconcile replicas and to ensure eventual consistency, as Bayou [110] did.

Medianode [107] is a distributed multimedia system supporting a replication mechanism based on the quality-of-service (QoS) characteristics of multimedia data and the availability of system resources. Medianode's replication mechanism offers a programming interface to update notification and update propagation and provides options such as update swapping, update dominating and update merging for conflict resolution.

Roma [146] is a centralized personal metadata service that stores information on each user file, such as names, location, timestamps and keywords, on behalf of mobility-aware applications. Synchronization agents help to ensure that the most up-to-date version of any document is available to the user on the storage devices he is currently using. Essentially, Roma is an Internet-based metadata management system. It does not deal with disconnection of portable devices and network partitions.

File hoarding under NFS and Linux [64] uses caches for hoarding files. Least Recently Used (LRU) is used for replacement policy.

Transaction logs are used during disconnection. A file update request will be logged to the transaction log before an actual copy of the file is updated. The transaction log is filename-based. It also records the file modification time and the attribute change time. In the implementation, the updates are appended to the log to avoid re-reading. In reconnection, multiple updates to the same file are ignored where possible. In addition, the last modification time of the file at the server is kept in a data structure for the local copy, which is later used to detect file modification conflicts. To deal with log overflow, redundant parts of the log file, such as records of multiple updates to the same file, are discarded every four hours. Periodic compression of the log file can be performed as well.

In resolving inconsistencies, renaming a file is regarded as deleting the file and creating a new file. This does not respect the user's conceptual view, nor does it reflect the actual implementation semantics of the renaming operation in Linux.

Unified caches [63] are designed to cache recently accessed files from heterogeneous servers. LRU is used as the hoarding policy. Moreover, the cache is visible as a logical device at the client side, which offers experienced users a lot of control over the cache content. Logs are used during disconnection to record data accesses.

On reintegration, logs are examined to detect conflicts. Write/write conflicts and read/write conflicts are reported to users. Renaming and deletion are not handled.

Globule [114] is a platform that automates replications of web documents on a world-wide scale, to reduce user-perceived latency and wide-area network traffic. In Globule, a web document is modelled as a physically distributed object whose state is replicated across the Internet.

Aspects related to replication, such as replication strategies and consistency management, are encapsulated inside objects. In Globule, logs are used to record document requests and are transmitted to master sites. Logs are used to evaluate replication strategies. The "best" policy can be chosen and dynamically applied to documents [115]. Globule is largely designed to serve read accesses to web documents. Its documents-as-objects model allows fine-tuning of data consistency on a document basis. Document objects in Globule should implement their own mechanisms to tackle network partitioning and device disconnection.

Other approaches and systems. An extension of the optimistic method, *commit-after*, is used in a disconnected database system [111]. In this approach, transactions can be locally committed during disconnection. Locally committed transactions release all locks and resources on the client. They may be rolled back on reintegration and must commit on the server to globally commit.

Disconnected updates on shared objects are recorded in semantic logs [135]. An update is described by a precondition, an operation and a postcondition. Precondition and postcondition capture the outcome of data synchronization, using first-order logic.

A hybrid method which controls database consistency is described in [94]. The method chooses an appropriate mechanism between token and optimistic methods based on the probability that transactions occur and the duration of disconnection time.

Locker rental service is a new type of Internet service providing users with data availability when mobile devices are disconnected [168]. In this approach, an agent represents and works for a user in a mobile device, keeping an amount of data for short periods of time. A user can contract the locker rental service to upload data for a long time. However, it is not clear how user data are kept consistent.

1.6.3 File synchronizers

File synchronizers are user-level programs, such as Briefcase of Windows systems, IntelliSync [119], Peacemaker Pro [27], SyncTalk [54], and QuikSync [69]. Those tools are used to manage files and their copies stored on different devices. They compare and synchronize file contents, according to pathname and last modification of files.

These tools are handy when the number of files is small, the directories in question are straightforward, and only a few devices are involved. They become hard to use in the case of large amounts of complex data. When using such tools, file and directory conflicts must often be manually repaired.

The Unison File Synchronizer [112] stores a kind of *archive* between synchronization. Archives describe the last synchronized states of replicas, different from logs that store disconnected update. Data synchronization of Unison is still vulnerable. In Unison, renaming a file has the same effect as that of the combination of deleting a file and creating a new file. This treatment violates the implementation semantics of file systems, that is, renaming a file does not change the internal index number (inode in Unix) of the file. Unison does not properly handle hard links in Unix systems either.

Interestingly, in the development of Unison, predicate calculus was used to specify the behavior of Unison in [10, 113]. In this formal model, file systems are regarded as trees, instead of acyclic graphs. Moreover, synchronization is regarded to be fully state-based, instead of history-based (log-based). This explains the vulnerability of Unison.

Filesystem algebra [125] gives an algebraic specification of file synchronization. It provides options for combining several conflict-resolution policies into the specification of a file synchronizer. This approach derives largely from the formal model of Unison.

1.6.4 Peer-to-peer file sharing

Peer-to-peer (P2P) file-sharing systems enable users to directly share files, such as MP3, music, movies, software, games and videos, with others without the need for a central file server. Examples are Gnutella [108], KaZaA [136], Freenet [24], Pastry [131] and OceanStore [78].

A network analysis shows that 70% of system users only download files without uploading any [3]. Moreover, according to a recent study focusing on user's perspectives on P2P systems [90], there is no evidence or tendency to suggest that people are willing to share their personal files, such as personal digital photos, in such systems. In the first place, this might be due to privacy concerns. P2P systems have the nature of anonymity, implying that it is impossible or very difficult for an outside observer to ascertain who has produced a file and who has examined it. To avoid risking personal documents being misused, people are unwilling to share personal files in such systems. Secondly, personal files need to have metadata so that they can be searched in such systems. As far as music and movie files are concerned, people need often not worry about this issue, thanks to the existence of several online databases such as the Gracenote music recognition service CDDB [50] and the Internet Movie Database [68]. As for personal files such as digital photos, however, people have to invest time and effort in associating metadata with the files. Therefore, it is not yet known to what extent people would exploit such systems to share personal files.

P2P systems have to achieve several goals that are difficult to achieve in traditional environments, such as *massive scalability.* A P2P system should work well with thousands, millions, or even billions of clients and files. Distributed systems in home environments need not address massive scalability. In a home environment, the number of devices and the number of files do not grow that much. It is outside the focus of this thesis to address such issues.

1.6.5 Summary

Most optimistically replicated systems have been built by extending or modifying existing systems. Their solutions are often restricted by the underlying platform and are often proprietary. A lot of systems have been developed to tackle conflicting updates. Each solution solves only a few problems. Some systems tackle update/update, update/rename, update/delete conflicts while others deal only with update/update conflicts.

In optimistically replicated systems, updates are recorded in logs, such as version vectors in Ficus and Rumor, resolution logs in Coda, operation logs in AFS, operations logs in Rover, write logs in Bayou, transaction logs in NFS, and logs in the unified cache system. This turns out to be common practice for achieving high-level consistency.

Log optimization is used to reduce storage spaces in storing updates, reduce network traffic in update propagation, and reduce the risk of conflicts. Log optimization is usually done in empirical manners in Ficus, Coda, AFS, Rover and Bayou. No formal guarantee is established to ensure that log optimization will not harm log replay and result in loss of information needed in conflict resolution.

Resolving conflicts is user-dependent. Several systems only provide conflict notification facilities and require user interaction. In the case of systems without logs, data synchronization is often done by comparing pathname and last-modification time of files, such as PFS, NFS, the unified cache system and file synchronizers. Several systems attempt to automatically resolve inconsistencies. Logs are replayed in Ficus, Rumor, Coda, Rover, Bayou. In addition, semantic rules are defined, such as the *no lost update* rule in Ficus. Nevertheless, those systems tend to inform users of the changes and allow users to remedy any errors.

1.7 Contribution of this thesis

Data inconsistencies introduced by disconnected updates are formally analyzed in this thesis. The analysis compiles all the known conflicts and discusses newly discovered conflicts. The resulting list of different types of inconsistencies can be used as a guideline in system design.

A new type of logs, *characteristic entry logs* [122], has been designed. Characteristic-entry logs record only the most recent accesses of each operation type of a data item. The compactness of such logs reduces storage space and network traffic in update propagation. A formal model of characteristic entry logs is presented.

Data synchronization is formalized. Semantic rules used in data synchronization are formally specified. The formal specifications provide a rigorous and unambiguous understanding of those rules. They can be used for testing and system verification.

Moreover, it is formally proven that the way that characteristic entry logs are constructed does not result in loss of information that is needed in data synchronization. This gives a formal justification that applying characteristic entry logs in real systems is safe.

A prototype system, the *MemorySafe* system [121], has been built as the underlying data management system of the Phenom demonstration applications. MemorySafe is a distributed data management system designed for home environments. MemorySafe provides flexible ways of managing multimedia data objects, by exploiting acyclic connected graphs as the data structure of the system. MemorySafe is capable of handling disconnection and provides a solution to data synchronization on device reconnection. In the MemorySafe system, applications can be built without awareness of data replication. If necessary, applications can be involved in data replication and data synchronization.

1.8 Structure of this thesis

This thesis is organized as follows. The first part discusses the background of the research, analyzes the problem and provides a survey of related work. In Chapter 2, several user scenarios involving disconnected updates are introduced. Next, the consequences of disconnected updates are discussed, in both informal and formal ways.

In the second part, an formal model of data synchronization in an HDMS is presented. In Chapter 4 an HDMS system is first formally described. Next, consistency requirements recording replicated data are specified. In the formal development of data synchronization, a formal model of characteristic-entry logs is presented. Moreover, the relation between characteristic-entry logs and normal logs is analyzed in a formal sense. In Chapter 5 it is proven that characteristic-entry logs can be used in resolving data inconsistencies introduced by disconnected updates. In this chapter, semantic rules used in data synchronization are first formalized. Then, the soundness of using characteristic-entry logs is proven when applying those rules.

In the third part, the system design, implementation and applications of an HDMS, the MemorySafe system, are presented. Chapter 6 presents the design and implementation of the system. Chapter 7 shows how data synchronization is addressed in the MemorySafe system and how characteristic-entry logs can be useful in the MemorySafe system and other similar systems. Several applications built on top of the MemorySafe system are presented in Chapter 8. Chapter 9 illustrates an application of a mathematical theory to photo-related applications. This application is also built on top of the MemorySafe system. Chapter 10 concludes this thesis.

2. Disconnected updates

Disconnected updates are data modifications that are made while devices are disconnected. In this chapter, several scenarios of disconnected updates will be described. Data inconsistencies that are introduced by disconnected updates will be analyzed. To resolve inconsistencies, it is necessary to introduce persistent immutable logical identities and access histories of data objects.

2.1 Scenarios

Disconnected updates occur both in home environments and in office environments. The scenarios presented in this section have been distilled from daily activities in both environments.

2.1.1 Managing digital photos at home

In home environments, people have all kinds of digital assets, such as digital photos, movies, and MP3 files. Such data are not only stored in stationary devices at home, but also copied to portable devices that people carry with them. Modifying one copy of a file without applying the same modification to the other copies may introduce inconsistencies. Typically, people manually manage the consistency between files and copies. Below are several scenarios excerpted from the Phenom Scenario Description [33, 34], illustrating how people manage digital photos.

Browsing photos at home. A user, whose name is Clair, has bought a digital photo-frame (Sepia) to display her photos in the living room at home. The Sepia discovers and connects to an existing home server containing digital photos. A photo browsing GUI is shown on the Sepia's touch screen, allowing Clair to browse her digital photo albums.

The Sepia discovers displaying devices, such as TV screens. The displays that are available in the living room are represented by pictorial screen icons on the Sepia's GUI. By dragging a photo thumbnail and dropping it onto a screen icon, Clair can show her photos on any screen in the vicinity. Figure 2.1 illustrates this scenario.

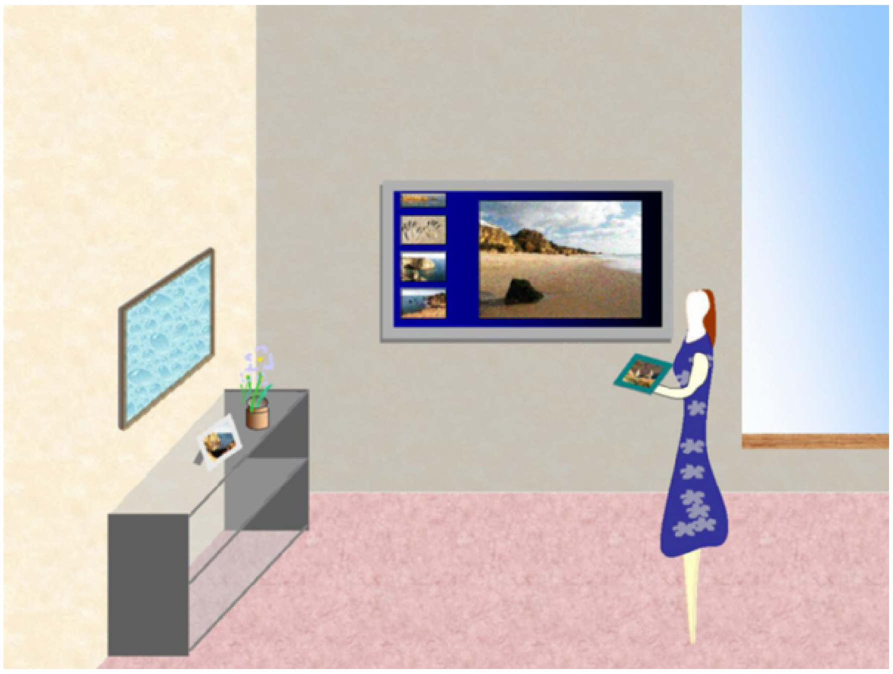


Figure 2.1. Browsing digital photos at home. A user can browse personal photo albums by using a portable device. She can also show photos on any displaying devices in the vicinity.

In addition, Clair uses souvenir objects to retrieve photos. First, she picks up a souvenir, say a palm tree souvenir which she bought on vacation, and puts it on a table. The souvenir is detected and an icon representing the souvenir is displayed on the Sepia GUI. By dragging a photo thumbnail from an album and dropping it onto the souvenir icon, Clair associates with the palm tree souvenir a collection of photos that were taken during her holidays. Next time when Clair puts the palm tree souvenir on the table, all the photos associated with that souvenir will be displayed on the Sepia GUI.

Exchanging photos on the move. Clair stores her favorite photos in her Sepia and takes the device along to visit her friend Mark. Before leaving home, Clair drags some photo thumbnails and drops them on a “backpack” icon on Sepia’s GUI. The backpack icon represents Sepia’s local storage. In this way, Clair copies photos from the home server to her Sepia.

When Clair visits Mark, her Sepia discovers Mark’s Sepia in the vicinity. A screen icon representing Mark’s Sepia appears on the GUI of Clair’s Sepia. Likewise, a screen icon representing Clair’s Sepia appears on the GUI of Mark’s Sepia.

To send a photo to Clair, Mark simply drags the thumbnail of the photo and drops it onto the screen icon representing Clair’s Sepia. Then, the photo

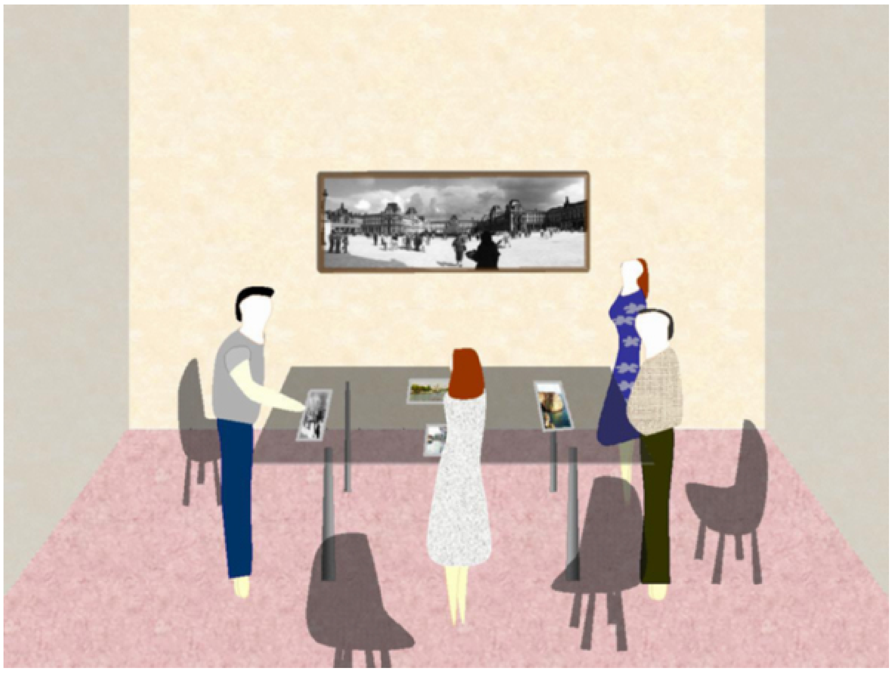


Figure 2.2. Sharing digital photos at home. A user can exchange personal photos with others, by using portable devices or, in this case, an interactive table with a display.

will appear on the screen of Clair's Sepia. To make a copy of the photo, Clair simply drags the received photo and drops it onto the backpack icon on the GUI of her Sepia. Sometimes Clair also annotates photos, for example, entering where and when photos were taken and with whom. Figure 2.2 illustrates this scenario.

Synchronizing photos at home. After returning home, Clair's Sepia synchronizes its content with the home server. All the new photos stored in Sepia are automatically uploaded to the home server. Modifications to the photos that are stored in Sepia are applied to the copies of those photos stored on the home server, if the copies are available.

Technical discussion. In home environments, digital photos are usually stored as files in computers. They are organized in directories, hierarchical structures provided by file systems of the computers. To manage photo files, people can use built-in file managing tools, such as Windows Explorer of Windows systems and File Manager of Unix systems. They can also work with software, such as ACDSee [2], Ulead Photo Explorer [161] and FlipAlbum [40]. In either way, photo files are accessed using filenames or pathnames in file systems.

When managing photos, people can view and modify photos. They can not only change the presentation of photos, such as orientation, color scheme and size, but also the metadata of photos, such as when, where and with whom the photos were taken. To organize photos, people have options of adding a photo to a directory, deleting a photo from a directory, copying and moving photos between directories, and renaming a photo in a directory. They can also make slide shows of sub-collections of photos for guest visits. Often, people have to copy photo files from one device to another.[1] Managing the consistency of the files and the copies is not an easy task when devices are disconnected. It will be shown in Section 2.4 that filename-based access is vulnerable to disconnected updates.

2.1.2 A businessman's way of working

The following scenarios demonstrate a businessman's way of working in the near future. The scenarios have been excerpted from the SyncML usage models [147].

The new mobile. A businessman uses the calendar and contacts on the company server. Then the businessman buys a new mobile. The businessman uses the calendar and the contacts on the server to synchronize his new mobile.

Company data sync. The businessman is traveling and meets several other people. During his trip the businessman arranges several future meetings. At the office the businessman's secretary does the same. The businessman saves his future meetings in his calendar. The secretary saves them on the server. Later at the hotel the businessman connects to the company server and synchronizes calendar data. One meeting in the businessman's calendar conflicts with one on the server. The businessman decides which meeting to keep and synchronizes both calendars.

Local sync. The businessman attends a meeting and a slide-set is presented in collaboration. The slide-set is synchronized with all attendants. The next meeting is arranged. One person collects all information and synchronizes with everyone at the end. The businessman synchronizes this information to his company database.

Mass Sync. The businessman arrives at the office and turns on his computer. His hand-held computer and his PC have Bluetooth and synchronize automatically. The businessman forgets to read his e-mail. At home he can still read messages from his hand-held computer.

[1] After copying a file f from a device S to a device T, the file created on T is called a *copy* of f.

Web-calendar. During his business trip, the businessman makes a lot of appointments. He synchronizes these appointments with his web calendar. The businessman suddenly remembers that he has to change something. He makes the change and synchronizes just this single item with one button. At home the businessman's son wants to inform his father of the Sunday football game. The son adds the game to the calendar. The change is automatically added to his father's mobile which alerts the father of the change.

Technical discussion. In office environments, calendar and contact data, emails and business information are often stored in database systems. The user has fewer operations to manipulate data in such systems than in file systems. Take the calendar application as an example. Each appointment in the calendar is treated as a uniquely identified record in a database. Thus, addition, editing and removal of an appointment are treated as insertion, modification and deletion of a record in the database, respectively.

As described in the scenarios, data are often replicated to improve data availability in database systems, especially for mobile computing applications.[2] Conflicting updates made while portable devices are disconnected need to be repaired by the user. Inconsistencies that are to be resolved will be discussed in Section 2.4.

2.2 Scenario analysis

In the photo browsing scenarios discussed in Section 2.1.1, the user can not only modify the photos, but can also modify albums by adding, removing or reordering photos. In data synchronization, both photos and albums need to be handled. Digital photos can be regarded as atomic data objects while albums are regarded as structured data objects. In this type of applications, the following operations are of relevance to disconnected updates.

- Read. Retrieve the stored information of a data object. For example, retrieve a photo or an album.
- Update. Store new information in an atomic data object. For example, crop a photo, scale a photo or change the annotation of a photo.
- Create. Create a data object in a structured data object. For example, add a photo to an album or associate a photo with a souvenir.
- Delete. Remove a data object from a structured data object. For example, remove a photo from an album.

[2] After a data item d has been replicated from a device S to a device T, the data item created on T is called a *replica* of d. Here, copying and replicating are regarded as distinct operations for migrating data between devices. Both operations have the effect of creating a data item on a target device, which has the same content as the original data item. However, copying is performed by the user while replicating is done by system administrators. Replication is transparent to the user.

- Rename. Change the reference name of a data object in a structured data object. For example, change a filename of a photo.

In the agenda scenarios discussed in Section 2.1.2, the businessman and his secretary modify the meeting information of the same time slot on the mobile and the server, respectively. A time slot in the businessman's agenda can be regarded as an atomic data object. When the business synchronizes the mobile with the server, conflicts can be detected by comparing the meeting information of the time slot. With this type of applications, the following operations are relevant to disconnected updates:

- Read. Retrieve the stored information of an atomic data object. For example, retrieve a meeting information of a time slot in an agenda.
- Update. Store new information in an atomic data object. For example, put a new meeting appointment in a time lot in the agenda. A meeting can be deleted by setting empty information in the time lot.

Note that the operations appearing in the agenda application will also appear in the photo application. In the rest of this section, analysis of disconnected updates will focus on photos and albums in the photo browsing scenarios.

2.2.1 Conflicts

Disconnected updates imply the following problems in the scenarios presented in Section 2.1.1. Those problems have been recognized and solved in many systems such as Ficus [126] and Coda [74].

- *Update two copies of a photo at the same time.*
  In the photo browsing scenario, Clair may change the annotation of a photo while visiting Mark. A copy of the photo may also be annotated by one of Clair's relatives, which would result in a conflicting disconnected update. In the case of such a conflict, the user must decide which update should be kept.
- *Update one copy of a data item while deleting another copy.*
  In the photo browsing scenario, Clair may delete a photo from one album on her Sepia while a copy of the photo at home is modified by one of Clair's relatives. In synchronization, it is hard to decide whether the copy on Clair's Sepia should be restored and updated with the new data of the copy at home or the copy at home should be deleted.
- *Update one copy of a data item while renaming another one.*
  In the photo browsing scenario, Clair may resize one copy of a photo on her Sepia while at home another copy of the same photo is renamed. One consequence of such renaming operations is that applications depending on the name of the photo won't function after the rename operation.

Table 2.1 lists all potential conflicting disconnected updates and discusses consequences of conflicts. The main findings are as follows.

Table 2.1. Checklist of disconnected updates on two copies of a structured data item. Assume two copies of a data item are involved. All combinations of disconnected updates on the copies are enumerated. Note that only the most recent operations on the data copies are considered.

Copy A	Copy B	Consequences	Negative effects in synchronization
read	read	-[a]	
read	update	-	
read	delete	-	
read	create	-	
read	rename	-	
update	update	Conflicting[b]	Determining which update to use will depend on the semantics of the operations. Ordering them by time does not always make sense, especially when updates were made by different users. One update is lost anyway in synchronization.
update	delete	Conflicting	The update would not be visible on the copy B, as it has already been deleted. In synchronization, the file can not be removed, when the deletion is desired.
update	create	Conflicting*	This happens after the copy B has been deleted; a file is created using the name of B. The new file has a content completely different from the copy A. Synchronization makes no sense in this case.
update	rename	Conflicting	After the renaming, the copy B has lost its identity. Applications dependent on the name will cease to function. This is called *identity loss*. When the rename operation is applied on the copy A, there may already be a file with the new name of the copy B. This is known as a *name clash*.
delete	delete	-	
delete	create	Conflicting*	Similar to update/create.
delete	rename	Conflicting	The renaming will not be visible on the copy A, as it has already been deleted. If the no-update-loss rule is applied, the file will never be deleted. Renaming causes identity loss.
create	create	Conflicting*	Similar to update/create. Name clashes.
create	rename	Conflicting*	Similar to update/create. Identity loss and name clashes may occur.
rename	rename	Conflicting*	The same as update/rename.

[a] No conflicts.
[b] Potential conflicts may occur.
* A conflict that is discovered in this analysis for the first time.

- Ordering disconnected updates is not always a reliable way of resolving conflicts. Two updates may have been performed by two users on different devices. Ordering them makes no sense, since they are actually unrelated.
- Resolving deletion-related conflicts often requires semantic knowledge to avoid data loss. Using the no-update-loss rule makes it hard to delete a data item.
- The name of a file is often used to capture the identity of the data stored in the file. After a file has been rename, the data stored in the file loses its identity. Applications that depend on the name of the file cease to work.
- Newly created files on different devices have different identities, even though their names happen to be same.
- In data synchronization, renaming or creation operations can introduce name clashes.

In the subsequent sections, a concrete example of structured data objects will be used, namely file systems, instead of only working on the abstraction level. The main reason for this approach is that most existing systems and tools handling disconnected updates are built on file systems. Thus, analyzing disconnected updates on file systems will not only provide insight into the problems involved in disconnected updates on structured data objects, but will also result in an overview of the state-of-the-art of data synchronization.

2.3 Basic definitions

A file system contains files storing data objects, where pathnames are used for locating files in the system. To avoid confusion, a definition of file systems is given here.

Definition 2.3.1. *A* file system *is a mapping between pathnames and data objects.*

Note that in a tree-like hierarchy, a pathname of a node is the concatenation of the names of nodes in the path that runs from the root node to the given node. Directories are not directly modelled by this definition. Empty directories are not considered either.

The hierarchy of a file system is reflected in pathnames. It is possible to construct the hierarchy of a file system by using valid pathnames of the system. Here, treating file systems as mappings between pathnames and data objects prevents the risk of the introduction of recursive data types when formalizing file systems.

Usually there will be only one path from the root to any node in a file system. In a DOS/Windows file system, this means a file can only be accessed by exactly one pathname, regardless of link files. In a Unix file system, however, a file can be located by multiple pathnames, by using hard links. According to the above-stated definition of file systems, both cases are allowed here. As far as non-hierarchically-organized data are concerned, their "hierarchy" is simply a flat set, which can also be modelled using the definition.

Below file systems are formally defined using the Z notation. To model file systems, two basic data types are introduced. *DATA* denotes data objects that can be stored in file systems. *PNAME* denotes all pathnames that can be used for locating files in file systems.

$$[DATA, PNAME].$$

A file system is modelled as a relation between pathnames and data objects. This relation should be functional, since a pathname can only refer to one file at a time in a system. Therefore, a file system is modelled as a partial function mapping pathnames to data objects in Z.[3] *FS* denotes the set of all file systems.

$$FS ::= PNAME \pfun DATA.$$

Figure 2.3 illustrates a file system on a file server, containing a "Favorites" directory . The directory contains two image files, "img1344.jpg" and "img1345.jpg". Let d_1 and d_2, where $d_1, d_2 \in DATA$, denote the contents of the files on the server, respectively. fs_s models the state of the file server.

$$fs_s = \{\text{``}Favorites\backslash img1344.jpg\text{''} \mapsto d_1, \text{``}Favorites\backslash img1345.jpg\text{''} \mapsto d_2\}.$$

[3] Alternatively, file systems can be regarded as directories. A directory is modelled as a mapping between filenames and subdirectories. The subdirectories can be files or directories. Mathematically, the universe of directories is defined as a recursive free type *DIR* in Z, as follows.

$$DIR ::= \langle\!\langle FNAME \pfun (DIR \cup DATA) \rangle\!\rangle.$$

where *FNAME* is a collection of file names. Correspondingly, operations on file systems should be defined recursively. For example, retrieving a data object by a pathname can be defined as follows.

```
$read : DIR \times PATHNAME \rightarrow DATA \cup \{\perp\}$
--------------------------------------------------------
$read = \lambda\, d : DIR;\ p : PATHNAME \bullet$
    **if** $p = \langle\rangle \vee head(p) \notin \operatorname{dom} d$ **then** $\perp$
    **else if** $d(head(p)) \in DATA \wedge tail(p) = \langle\rangle$ **then** $d(head(p))$
        **else if** $d(head(p)) \in DATA \wedge tail(p) \neq \langle\rangle$ **then** $\perp$
            **else** $read(d(head(p)), tail(p))$
```

Note that $\perp$ stands for undefinedness. Also, a pathname is modelled as a sequence of file names, instead of as an element of the type *PNAME*.

$$PATHNAME ::= \operatorname{seq} FNAME.$$

Such comprehensive specifications may needlessly complicate the analysis of disconnected updates. They are therefore avoided whenever possible. Hierarchical pathnames will therefore be used in this thesis to capture hierarchical structures of file systems.

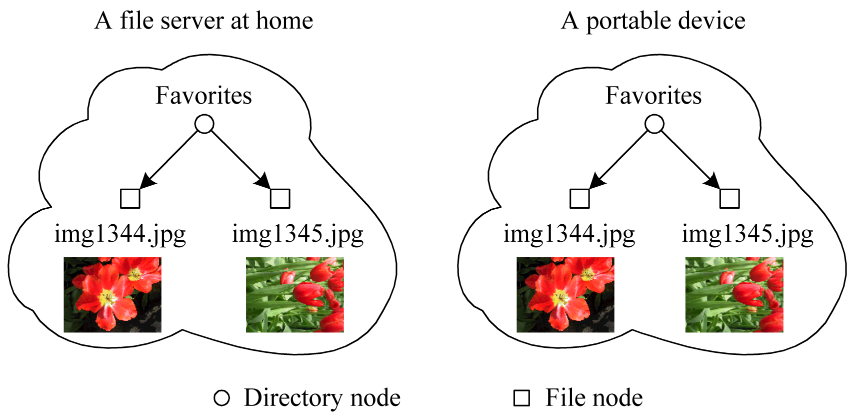


Figure 2.3. The directory "Favorite" on a file server at home and that on a portable device contain identical photos.

Note that "*Favorites*" is the name of the parent node of d_1 and "$img1344.jpg$" is the filename of d_1. In the formalization, complete pathnames are used to identify files. In Figure 2.3, d_1 is referred to by "$Favorites \backslash img1344.jpg$". The figure also shows a file system on a portable device. In the portable device, the "Favorites" directory is an identical copy of the "Favorites" directory in the server. The directory has the same contents as the one on the file server.

Definition 2.3.2. *The operations that can be performed on a file system are* create*,* update*,* rename*,* delete *and* read*.*

These operations have been abstracted from the scenario analysis described in Section 2.2. The create operation is used for creating a new file in a file system to store data. The update operation is meant for modifying the content of a file. The rename operation changes a pathname of a file. The delete operation removes a file and its data from a system. These operations will be formally defined using Z in the next section.

Files can be copied or replicated onto different devices. When devices are disconnected, files can be accessed independently of their copies. Disconnected updates are the modifications made on files and their copies when file systems are disconnected. With respect to the definitions of file systems and operations on a file system, disconnected updates can be any executions of create, update, rename and delete operations on files and their copies.

2.4 Formal analysis

Due to disconnected updates, replicated file systems can become inconsistent. A file and its copies may for example contain different data. The con-

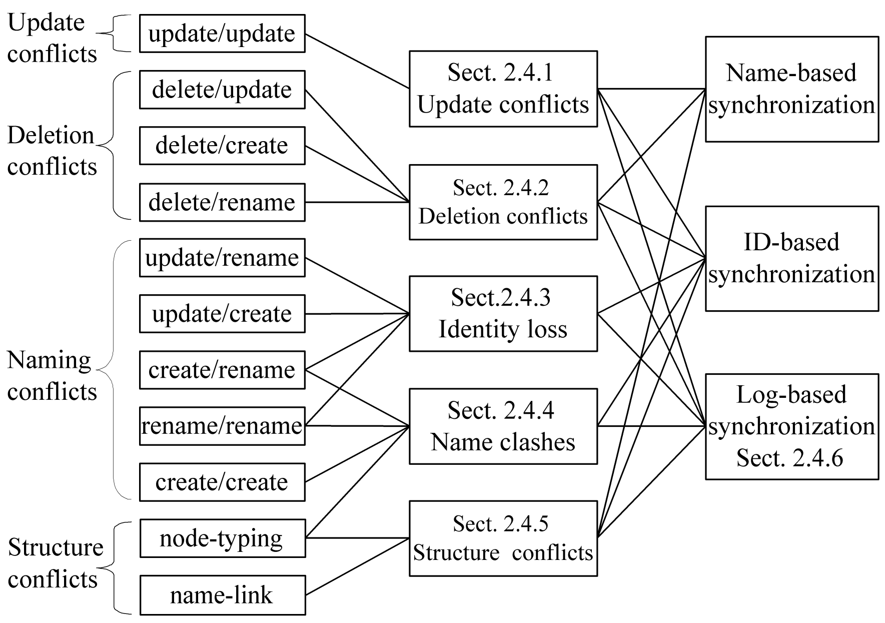


Figure 2.4. A pictorial overview of Section 2.4. The left part indicates where a specific type of disconnected updates is discussed. The right part shows which conflicts a specific synchronization method is able to address.

flicting disconnected updates in Table 2.1 have been grouped into three categories:*update conflicts*, *deletion conflicts*, and *naming conflicts*, as shown in Figure 2.4. In addition, *structure conflicts*, conflicts that are related to structures of file systems, will also be discussed.

Data synchronization resolves inconsistencies of disconnected updates. This process is also called *conflict resolution.* Existing techniques of data synchronization are based largely on pathnames, identifications and logs. Figure 2.4 indicates how each method addresses the above-mentioned conflicts.

2.4.1 Update conflicts

In file synchronizers, pathnames are used for identifying files and their copies across systems. Files or their copies can be updated in a disconnected manner.

Updating a file can be modelled by the *update* function as follows.

$$update : FS \times PNAME \times DATA \rightarrow FS$$

$$update = \lambda\, s : FS;\ p : PNAME;\ d : DATA \bullet$$
$$\quad \textbf{if}\ p \notin \operatorname{dom} s\ \textbf{then}\ s\ \textbf{else}\ s \oplus \{p \mapsto d\}$$

Updating a file p in a file system has the following effect on the system. If the pathname p is not defined in the system, then the update operation will not change the original system. Otherwise, the new data, denoted by d, will be used to replace the old data of p in the system.

For example, the file systems on the server and the portable device in Figure 2.3 are updated individually while they are disconnected. After the "*Favorites*\img1344.*jpg*" file has been updated, the file system on the server at home will have a new state fs'_s.

$$\begin{aligned} fs'_s &= update(fs_s, \text{“}Favorites\backslash img1344.jpg\text{”}, d_3) \\ &= \{\text{“}Favorites\backslash img1344.jpg\text{”} \mapsto d_3, \text{“}Favorites\backslash img1345.jpg\text{”} \mapsto d_2\}. \end{aligned}$$

Initially, the state of the file system on the portable device will be the same as that on the server.

$$fs_c = \{\text{“}Favorites\backslash img1344.jpg\text{”} \mapsto d_1, \text{“}Favorites\backslash img1345.jpg\text{”} \mapsto d_2\}.$$

After the "*Favorites*\img1344.*jpg*" file has been updated, the file system on the portable device will have a new state fs'_c.

$$\begin{aligned} fs'_c &= update(fs_c, \text{“}Favorites\backslash img1344.jpg\text{”}, d_4) \\ &= \{\text{“}Favorites\backslash img1344.jpg\text{”} \mapsto d_4, \text{“}Favorites\backslash img1345.jpg\text{”} \mapsto d_2\}. \end{aligned}$$

When synchronizing the server and the portable device, it will be found that the "*Favorites*\img1344.*jpg*" file has different data on the two devices. This is a so-called *update/update conflict.* (Note that update/update conflicts also occur in database systems where a record and its copies can be updated independently.) Such conflicts can be detected by checking the last-modification property of the files or performing a byte-wise data comparison of the files.

The last-modification property of files can be used to resolve such conflicts. The most recent modification wins in determining what data the files should contain after synchronization, which is called the "up-to-date" rule. This method requires that system clocks are synchronized.

Ordering two updates is not always a meaningful way of determining which update should be applied. If there is no cause-effect relation between the two updates, there will be no need to order updates at all. For example, disconnected updates that are performed by two users are often unrelated to each other. In this case, ordering them by time and selecting the most recent updates makes no sense.

To solve such conflicts, updates can sometimes be merged to form new versions, as a Concurrent Version System (CVS) does. In practice, update/update conflicts are often reported to the user and the user can determine what to do in data synchronization. The user has the best knowledge of the modifications. Propagating the most recent modification may result in loss of valuable data. User-conducted resolution can avoid errors in deciding which file contains the most recent data.

2.4.2 Deletion conflicts

In disconnected updates, a file can be updated in a file system, while its copy is removed from another file system. This is known as an *update/delete conflict*.

File deletion can be described by the *delete* function .

$$
\begin{array}{|l}
delete : FS \times PNAME \to FS \\
\hline
delete = \lambda\, s : FS;\ p : PNAME \bullet \\
\qquad \mathbf{if}\ p \notin \operatorname{dom} s\ \mathbf{then}\ s\ \mathbf{else}\ s \lhd (\operatorname{dom} s \setminus \{p\})
\end{array}
$$

When deleting a file from a file system, the system will remain untouched if the specified pathname has not yet been defined in the system. Otherwise, the mapping matching the specified pathname will be removed.

For example, let fs'_s be the state after the updating of the file system on the server, as described in Section 2.4.1. Let fs'_c be the state after the deletion of the "*Favorites*\img1344.*jpg*" file from the file system on the portable device.

$$
\begin{aligned}
fs'_c &= delete(fs_c, \text{“}Favorites\backslash img1344.jpg\text{”}) \\
&= \{\text{“}Favorites\backslash img1345.jpg\text{”} \mapsto d_2\}.
\end{aligned}
$$

When synchronizing fs'_s and fs'_c, an update/delete conflict will occur. The deleted file is regarded as a *file miss*.[4]

In non-log based synchronization, update/delete conflicts are not considered at all, since after a file has been deleted, a file system will no longer contain information on the deleted file.[5] This situation is often called *create/delete ambiguity* [52]. When comparing two file systems, one of which contains a copy of a file and the other does not, it is impossible to determine whether the file was newly created in the first system or has been removed from the second system. In non-log-based data synchronization, the deletion operation is ignored and the updated copy of the file is propagated. In the above-mentioned example, the mapping "*Favorites*\img1344.*jpg*" $\mapsto d_3$ is added to fs'_c after the synchronization.

In log-based data synchronization, the up-to-date rule, which is used to resolve update/update conflicts, is not often applied to update/delete conflicts. Instead, the updated file is often propagated to the device where the file was deleted, to avoid update loss. This is called *no lost update* semantics in the Ficus system [126]. The disadvantage of using this rule is that it is impossible to forcibly propagate the delete operation in the case of any updates on the same data.

[4] Note that update/delete conflicts may also occur in database systems. A record may be updated while its copy is removed. These two operations can be performed independently, resulting in an update/delete conflict.

[5] In Windows file systems, deleted files are moved to recycle bins for later recovery. However, from a programmer's point of view, such information is not available for synchronization.

With respect to the delete operation, a *delete/rename* conflict refers to the situation where a copy of a file was deleted at one site while another copy of the same file was renamed at another site. A *delete/create* conflict will occur if a copy of a file was deleted at one site while at another site a copy of the same file was deleted and afterwards the filename was used for a newly created file with new data. In non-log based synchronization, such conflicts are treated in the same way as update/delete conflicts. When introducing logs or identities in data synchronization, such conflicts can be dealt with as will be shown in the subsequent two sections.

2.4.3 Identity loss

The rename operation of file systems can make pathname-based data synchronization vulnerable, due to the fact that logical identities of files are lost after the renaming.

The rename operation of file systems can be formally modelled by the *rename* function as follows.

$$\begin{array}{|l} rename : FS \times PNAME \times PNAME \rightarrow FS \\ \hline rename = \lambda\, s : FS;\ p_1, p_2 : PNAME \bullet \\ \quad \textbf{if } p_1 \notin \operatorname{dom} s \vee p_2 \in \operatorname{dom} s \vee p_1 = p_2 \textbf{ then } s \\ \qquad \textbf{else } s \oplus \{p_2 \mapsto s(p_1)\} \lhd \{\operatorname{dom} s \cup \{p_2\} \setminus \{p_1\}\} \end{array}$$

Given a file system s, renaming a file from p_1 to p_2 has the following effect on s. If the pathname p_1 is not defined in the system, p_2 is already used in the system, or p_1 and p_2 are the same, then the *rename* operation will not change the original system. Otherwise, a new mapping will be added to the system, which will map p_2 to the original file of p_1, and p_1 will become undefined in the new system state.

Take the example in Figure 2.3. Initially, the file system on the portable device had the state fs_c.

$$fs_c = \{\text{“}Favorites\backslash img1344.jpg\text{”} \mapsto d_1, \text{“}Favorites\backslash img1345.jpg\text{”} \mapsto d_2\}.$$

During disconnection, the file d_1 is renamed to “*Favorites\flowers*”. The state of the file system at the portable device then becomes fs'_c.

$$\begin{aligned} fs'_c &= rename(fs_c, \text{“}Favorites\backslash img1344.jpg\text{”}, \text{“}Favorites\backslash flowers\text{”}) \\ &= \{\text{“}Favorites\backslash flowers\text{”} \mapsto d_1, \text{“}Favorites\backslash img1345.jpg\text{”} \mapsto d_2\}. \end{aligned}$$

When the portable device is reconnected to the server, it is discovered that the “*Favorites\img1344.jpg*” file exists only in the server while the “*Favorites\flowers*” file exists only on the portable device. Although both files contain the same data, they are treated as different files in data synchronization. Renaming a file causes the loss of its logical identity, which is called *identity loss*.

In pathname-based data synchronization, "$\mathit{Favorites}\backslash \mathit{img}1344.\mathit{jpg}$" on the server will be copied to the portable device and "$\mathit{Favorites}\backslash \mathit{flowers}$" on the portable device will be copied to the server. After synchronization, the file systems on both devices should look as follows.

$$\{\text{“}\mathit{Favorites}\backslash \mathit{flowers}\text{”} \mapsto d_1, \text{“}\mathit{Favorites}\backslash \mathit{img}1344.\mathit{jpg}\text{”} \mapsto d_1, \\ \text{“}\mathit{Favorites}\backslash \mathit{img}1345.\mathit{jpg}\text{”} \mapsto d_2\}.$$

This result is not entirely what a user would expect. A user may think that the name of the "$\mathit{Favorites}\backslash \mathit{img}1344.\mathit{jpg}$" file on the server should be changed to "$\mathit{Favorites}\backslash \mathit{flowers}$".

From a user's point of view, renaming a file does not change the content of the file and, conceptually, should leave the *identity* of the file unchanged. In a file system, renaming a file is implemented in such a way that the internal identification of the file remains unchanged. Unfortunately, pathname-based data synchronization is not capable of properly handling renamed files. Without careful treatment, identity loss causes the problem that applications depending on pathnames fail to work. Moreover, it becomes difficult to check whether two files are copies of the same digital photo.

2.4.4 Name clashes

To deal with the vulnerability of pathname-based data synchronization, unique identifications of files are used in data synchronization in the Ficus, Coda and Rover systems. In database systems, global unique identifiers (GUIDs) can be assigned to records. The introduction of unique identifications helps to resolve the identity loss problem. User intervention is still required in resolving several renaming-related conflicts.

A two-layered naming mechanism is used in the Ficus, Coda and Rover systems. In the application layer, pathnames are used for accessing files. In the system kernel, pathnames are translated into unique identifications, which are used for fast location of data on physical storage media. Such unique identifications are assigned to new files and are immutable. Extensions of inodes in Unix systems are often used to form identifications.

To model a file system using a two-layered naming mechanism, a basic data type ID is introduced.

$$[\mathit{ID}].$$

ID denotes a set of unique identifiers. For example, inodes in a Unix file system can be such identifiers.

A file system, hence, consists of two mappings, one from PNAME to ID, the other from ID to DATA.

$$\mathit{FSi} ::= \{(\mathit{fn}, \mathit{fi}) : (\mathit{PNAME} \rightarrowtail\!\!\!\!\!+ \, \mathit{ID}) \times (\mathit{ID} \rightarrowtail\!\!\!\!\!+ \, \mathit{DATA}) \mid \operatorname{ran} \mathit{fn} = \operatorname{dom} \mathit{fi}\}.$$

Given such a file system, (fn, fi), fn is called the *naming function* and fi is called the *retrieving function*. The range of fn should coincide with the domain of fi. This requirement is needed to avoid two erroneous situations: an assigned identifier that is not referenced by any pathname, and an assigned pathname that is not related to any data.

Creating a file in a file system can be modelled by the *create* function.

$$\mathit{create} : \mathit{FSi} \times \mathit{PNAME} \times \mathit{DATA} \times \mathit{ID} \rightarrow \mathit{FSi}$$

$$\begin{aligned}&\mathit{create} = \lambda(fn, fi) : \mathit{FSi};\ p : \mathit{PNAME};\ d : \mathit{DATA};\ i : \mathit{ID} \bullet\\ &\quad \textbf{if } p \in \operatorname{dom} fn \vee i \in \operatorname{dom} fi \textbf{ then } (fn, fi)\\ &\qquad \textbf{else } (fn \oplus \{p \mapsto i\}, fi \oplus \{i \mapsto d\})\end{aligned}$$

To create a file p in a system s, a free identifier should first be obtained. When there is no such free identifier, or p has already been used in the naming function of s, then the create operation will not change the file system. If a free identifier i exists and p has not yet been defined in the naming function of s, then a mapping from p to i is added to the naming function, while a mapping from i to the data of p is added to the retrieving function.

A modified version of the *rename* function is defined here, which works for a file system with a two-layered naming mechanism. This definition manifests the semantics of a renaming operation in file systems, namely keeping the identity of the file unchanged.

$$\mathit{rename} : \mathit{FSi} \times \mathit{PNAME} \times \mathit{PNAME} \rightarrow \mathit{FSi}$$

$$\begin{aligned}&\mathit{rename} = \lambda(fn, fi) : \mathit{FSi};\ p_1, p_2 : \mathit{PNAME} \bullet\\ &\quad \textbf{if } p_1 \notin \operatorname{dom} fn \vee p_2 \in \operatorname{dom} fn \vee p_1 = p_2 \textbf{ then } (fn, fi)\\ &\qquad \textbf{else } (fn \oplus \{p_2 \mapsto fn(p_1)\} \lhd \{\operatorname{dom} s \cup \{p_2\} \setminus \{p_1\}\}, fi)\end{aligned}$$

The rename function defined here is easily distinguishable from the one defined in Section 2.4.3 according to the context.

Take the example in Figure 2.3. Now, let's assume that the file systems in the figure have a two-layer naming mechanism. Let fsi_s denote the initial state of the file system on the server.

$$\begin{aligned}fsi_s = (&\{\text{“}\mathit{Favorites}\backslash \mathit{img}1344.\mathit{jpg}\text{”} \mapsto i_s^1, \text{“}\mathit{Favorites}\backslash \mathit{img}1345.\mathit{jpg}\text{”} \mapsto i_s^2\},\\ &\{i_s^1 \mapsto d_1, i_s^2 \mapsto d_2\}),\end{aligned}$$

where i_s^1, $i_s^2 \in \mathit{ID}$. Let fsi_c denote the initial state of the file system on the portable device.

$$\begin{aligned}fsi_c = (&\{\text{“}\mathit{Favorites}\backslash \mathit{img}1344.\mathit{jpg}\text{”} \mapsto i_c^1, \text{“}\mathit{Favorites}\backslash \mathit{img}1345.\mathit{jpg}\text{”} \mapsto i_c^2\},\\ &\{i_c^1 \mapsto d_1, i_c^2 \mapsto d_2\}),\end{aligned}$$

where i_c^1, $i_c^2 \in \mathit{ID}$. Initially the files have the same data on both sides.

Internal identifiers of files are propagated when files are replicated. In this example, both devices keep record of the identifiers of the files being replicated to each other. Formally, such mappings are modelled as elements of *IMAP*.

$$IMAP ::= ID \rightarrowtail \mathbb{P}\, ID.$$

For example, in Figure 2.3, the following mapping is available at the server side.

$$imap_s = \{i_s^1 \mapsto \{i_c^1\}, i_s^2 \mapsto \{i_c^2\}\}.$$

From this mapping, the server can tell whether a local file has a copy in another file system. If so, the identifier of a file from the system should be mapped to an internal identification of the local file system at the server. At the portable device, a similar mapping is available.

$$imap_c = \{i_c^1 \mapsto \{i_s^1\}, i_c^2 \mapsto \{i_s^2\}\}.$$

Consider a "create/rename" scenario. A new "*Favorites*\flowers" file is created on the server, while the "*Favorites\img*1344.*jpg*" file is renamed to "*Favorites\flowers*" on the portable device. The new state of the file system at the server is fsi_s'.

$$\begin{aligned} fsi_s' &= create(fsi_s, \text{"}Favorites\backslash flowers\text{"}, d_3, i_s^3) \\ &= (\{\text{"}Favorites\backslash img1344.jpg\text{"} \mapsto i_s^1, \text{"}Favorites\backslash img1345.jpg\text{"} \mapsto i_s^2, \\ &\qquad \text{"}Favorites\backslash flowers\text{"} \mapsto i_s^3\}, \{i_s^1 \mapsto d_1, i_s^2 \mapsto d_2, i_s^3 \mapsto d_3\}), \end{aligned}$$

where $d_3 \in DATA$, $i_s^3 \in ID$ and i_s^3 was not assigned before the creation. At the portable side, the new state of the file system is fsi_c'.

$$\begin{aligned} fsi_c' &= rename(fsi_c, \text{"}Favorites\backslash img1344.jpg\text{"}, \text{"}Favorites\backslash flowers\text{"}) \\ &= (\{\text{"}Favorites\backslash flowers\text{"} \mapsto i_c^1, \text{"}Favorites\backslash img1345.jpg\text{"} \mapsto i_c^2\}, \\ &\qquad \{i_c^1 \mapsto d_1, i_c^2 \mapsto d_2\}) \end{aligned}$$

Synchronizing fsi_s' and fsi_c' involves two steps. First, from $imap_s$, it's known that i_s^1 in fsi_s' corresponds to i_c^1 in fsi_c'. Since the file identified by i_c^1 has been renamed in the portable device, the original copy of the file in the server should also be renamed. Therefore, the state of the file system on the server, fsi_s'', should be as follows.

$$\begin{aligned} fsi_s'' &= rename(fsi_s', \text{"}Favorites\backslash img1344.jpg\text{"}, \text{"}Favorites\backslash flowers\text{"}) \\ &= fsi_s'. \end{aligned}$$

The rename operation fails to change the state of the file system, because in this example the pathname "*Favorites*\flowers" has already been used for a newly created file. This situation is called *name clash.* Second, the newly created file at the server should be propagated to the portable device. Therefore, the new state of the file system on the portable device, fsi_c'', should be as follows.

$$\begin{aligned} fsi''_c &= create(fsi'_c, \text{“}Favorites\backslash flowers\text{”}, d_3, i^3_c) \\ &= fsi'_c, \end{aligned}$$

where i^3_c is a fresh identifier on the portable device. The create operation leaves the state of the file system on the portable device unchanged, due to the fact that the pathname “*Favorites\flowers*” has already been used in a recent renaming operation, which is another example of *name clash*.

In practice, users are notified to resolve name clashes. They can manually modify the names of conflicting files. Alternatively, systems can create new files containing contents of the conflicting files. In either case, applications depending on pathnames may fail to work after synchronization.

Different from treating files as objects with identities, synchronization can be performed according to filenames in this case. Contents of files are regarded as “properties” of filenames and what synchronization does is to make sure two filenames have the same contents. From this perspective, name clashes can be resolved by using a priori rules like propagating the most recent modification in the case of name conflicts. Incautiously applying such rules may incur data loss.

2.4.5 Structure conflicts

In data synchronization, structure conflicts refer to inconsistencies between structures of file systems. There are two types of structure conflicts: *name-link* conflicts and *node-typing* conflicts.

Name-link conflicts. One example of a name-link conflict is shown in Diagram (A) of Figure 2.5. On the file server at home, “*Favorites\img*1344.*jpg*” and “*Favorites\flowers*” refer to different files and the identifications of the files are distinct. On the portable device, however, both pathnames, “*Favorites\img*1344.*jpg*” and “*Favorites\flowers*”, refer to the same file.[6]

Attempting to resolve name-link conflicts is futile in the absence of additional information. For example, suppose the last modification of the “*Favorites\flowers*” file on the server is more recent than that on the portable device. First, synchronizing “*Favorites\img*1344.*jpg*” on the server and the portable device will leave the states of both devices untouched. Next, when synchronizing the “*Favorites\flowers*” file, the file on the server will be propagated to the portable device according to the assumption that the most recent modification was made on the file on the server. This operation will transfer the content of the “*Favorites\flowers*” file on the server to the portable device. Since both “*Favorites\flowers*” and “*Favorites\img*1344.*jpg*” are links to the same file on the portable device, they both refer to the data of the “*Favorites\flowers*” file on the server after propagation. Now a new inconsistency is introduced, in that “*Favorites*1344.*jpg*” refers to different contents on the server and on the portable device.

[6] In the Unix file system, such a situation can be created by using hard links. A file may have multiple hard links to it.

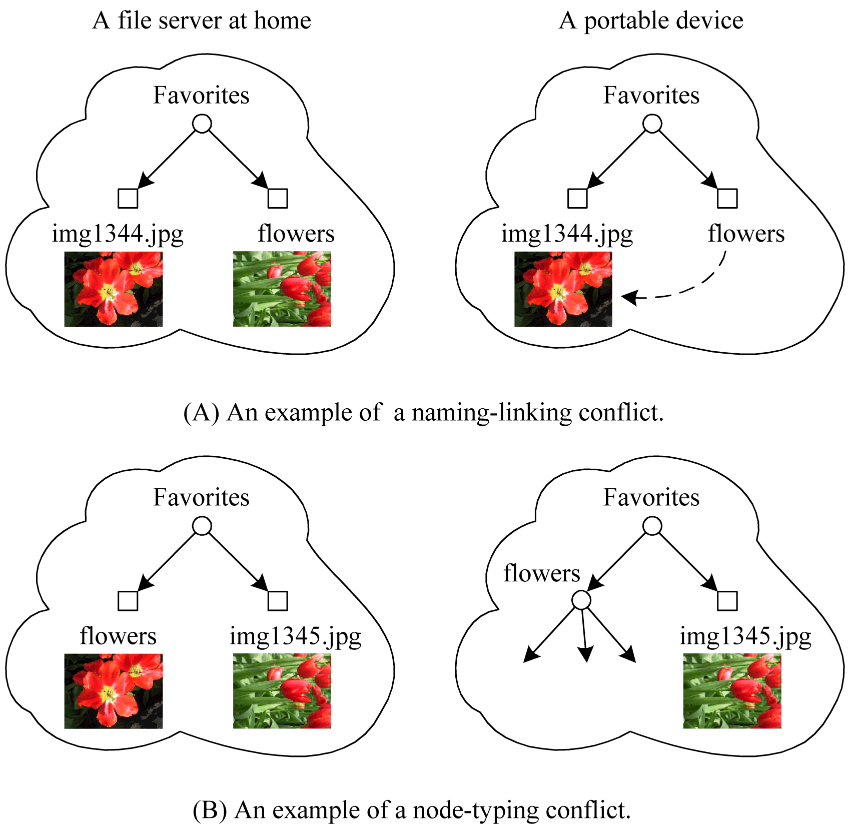


Figure 2.5. Examples of structure conflicts.

One way of dealing with such conflicts is to break the links and make a full copy of the file being linked. More decent approaches rely on internal structures of underlying file systems. In order to solve such conflicts, links should be distinguishable from filenames. Moreover, the internal identifications used for files can be used to check wether two filenames refer to the same file or not. These requirements are not satisfied in Unix file systems. In the first place, in a Unix file system a hard link is indistinguishable from a new file reference, which makes it almost impossible to resolve such conflicts in the Unix file system. Secondly, inodes, the internal identifications used for files, are recycled in a Unix file system: an inode can be used for different files at two instances of time. So, inodes are not reliable for identifying the identities of the files.

Node-typing conflicts. Diagram (B) of Figure 2.5 shows one example of a node-typing conflict. In the figure, the pathname "*Favorites\flowers*" refers

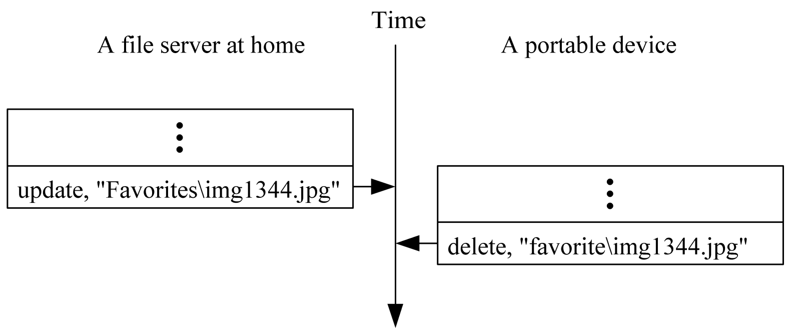

Time
A file server at home
A portable device
update, "Favorites\img1344.jpg"
delete, "favorite\img1344.jpg"

Figure 2.6. Logs are used to resolve an update/delete conflict. In the diagram, timestamps of log entries have been omitted and the time ordering of log entries is indicated by the time line.

to a file at the home server, while the same path points to a directory on the portable device.

In pathname-based or identification-based synchronization, such conflicts can be easily detected. Resolving them may introduce name clashes. To deal with such a conflict, one solution would be to delete either the file or the directory specified by the given pathname. In this case especially, choosing the file to overwrite the directory will remove all sub directories of the directory, causing data loss. For data safety concerns, a rather conservative and safe approach is to notify the user and to let the user decide what to do next. Alternatively, either the file or the directory can be automatically renamed to a new name, which would automate the conflict resolution. Applications depending on pathnames may fail to work after synchronization.

2.4.6 Log-based synchronization

Log-based data synchronization uses histories of data updates in data synchronization. Logging mechanisms are used to record disconnected updates. On reconnection, logs are used for rolling back system states to a consistent state before disconnection. Then, entries of logs from different devices are ordered, for example by time. This process is know as *serialization* in database systems[19, 23, 28]. The serialized entries are applied to all devices in a transactional manner, ensuring that at the end of the synchronization all devices have been modified in the same way. Log-based synchronization can be based on pathnames or identifications.

Log replay. Several types of conflicting disconnected updates can be resolved in log-based data synchronization, such as update/delete conflicts, update/rename conflicts and delete/rename conflicts.

Take the update/delete conflict in Section 2.4.2 as an example.[7] Initially the file server at home and the portable device have identical contents in the "*Favorites*" directory. During disconnection, the contents in the devices were modified. As shown in Figure 2.6, in the log of the server, there is one entry recording a modification of the "*Favorites*\img1344.*jpg*" file; in the log of the portable device, there is one entry of deletion of the "*Favorites\img*1344.*jpg*" file. Suppose that the update operation was performed before the delete operation.

In synchronization, the states of the server and the portable device are rolled back to the initial consistent state. When the serialized log entries are applied, the update operation will first be applied to both devices and then the delete operation will be applied. After the synchronization, the "*Favorites*\img1344.*jpg*" file will be removed from both devices. The server and the portable device will have then become consistent.

Next, assume the delete operation was performed before the update operation in the example shown in Figure 2.6. In synchronization, the delete operation will be applied to both devices. When the update operation is applied, it will be found that the file to be updated has just been deleted. The update operation is lost in synchronization. To tackle this problem, it is necessary to "look ahead" in the serialized log entries to check whether the file to be deleted will be updated (or renamed) as well. If so, the log entry containing the delete operation will be discarded.

Name clashes. Name clashes remain problematic in log-based synchronization. Take the create/rename conflict presented in Section 2.4.4. Initially the file server at home and the portable device have identical contents in the "*Favorites*" directory. During disconnection, a "*Favorites*\flowers" file was created on the server while on the portable device the "*Favorites\img*1344.*jpg*" file was renamed to "*Favorites\flowers*", as shown in Figure 2.7. Suppose that the rename operation was performed before the create operation.

In log-based data synchronization, the states of the server and the portable device are rolled back to the initial consistent state. Then, the rename operation is applied to both devices after serialization. When the second operation is applied, that is, the create operation, the application will fail, because the name "*Favorites*\flowers" already exists. To avoid such data loss, special treatments are needed, such as manual repair, or automatic renaming of filenames.

Technical limitations. The size of a log grows linearly with the number of performed accesses. The log size may become rather big if portable devices remain disconnected for a long time. Consequently, log-based synchronization has technical drawbacks. In the first place, rolling-back, serializing log entries, and applying serialized log entries are both time-consuming and computation-intensive operations. Especially transporting lengthy logs and applying serialization may slow down data synchronization, which is often time-critical:

[7] Log-based data synchronization will be formally studied in Chapters 4 and 5.

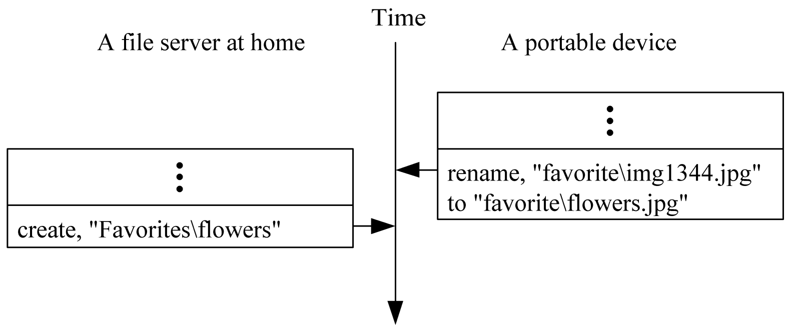

Time
A file server at home
A portable device
create, "Favorites\flowers"
rename, "favorite\img1344.jpg" to "favorite\flowers.jpg"

Figure 2.7. Name clashes can occur in log-based synchronization. In the diagram, timestamps of log entries have been omitted and the time ordering of log entries is indicated by the time line.

users are waiting for it to finish. Mainframe and desktop computers are suitable for such operations while portable devices, especially portable consumer electronics devices, might not be able to afford those expensive operations. Secondly, portable devices often have limited storage capacity. Although storage involved in keeping logs becomes less a primary concern than efficiency and correctness of using logs, storage constraints are still of relevance for miniaturized devices. In those devices, logs may overflow, causing information loss in log files.

2.4.7 Summary

The inconsistencies discussed in Sections 2.4.1 to 2.4.6 are summarized in Table 2.2. It is also indicated in the table how the inconsistencies are treated in pathname-based, identification-based and log-based data synchronization.

Generally speaking, pathname-based data synchronization performs static state-based data synchronization. It ignores the identity loss problem and the name clash problem completely. This has to do with the fact that pathname-based data synchronization is not data-oriented. This solution sacrifices the identities of data objects and is a reliable approach to disconnected updates. Furthermore, name-link conflicts cannot be detected and resolved. Node-typing conflicts can be detected. But resolving them often requires user involvement.

Identification-based data synchronization deals with update/update conflicts and deletion-related conflicts, in the same way as pathname-based synchronization. In dealing with deletion-related conflicts, this method regards a deleted file as file miss. In synchronization, the deleted file will be restored, which might be annoying to the user. Identification-based data synchronization addresses the identity loss problem. Unfortunately, resolving rename-

Table 2.2. Problematic disconnected updates in data synchronization. Assume two file systems, fs_1 and fs_2, and a pathname p. The "Problematic disconnected updates" column lists all combinations of operations that can be applied to fs_1 and fs_2 individually, introducing inconsistencies. The "Data synchronization" column indicates how different synchronization methods deal with those situations.

Problematic disconnected updates		Data synchronization		
		Name-based	ID-based	Log-based
update	update/update	solved(latest)[a]	solved(latest)	solved(L.R.)[b]
deletion	delete/update	file miss[c]	file miss	solved(L.R.)
	delete/create[d]	file miss	file miss	solved(L.R.)
	delete/rename	file miss	file miss	solved(L.R.)
naming	update/rename	identity loss[e]	name clash[f]	name clash
	update/create[g]	identity loss	name clash	name clash
	create/create	identity loss	name clash	name clash
	create/rename	identity loss	name clash	name clash
	rename/rename	identity loss	name clash	name clash
structure	name-link*	unsolved	solved	unsolved
	node-typing	unsolved	name clash	unsolved

[a] This conflict can be resolved by propagating the latest update to both sides, though one update will be lost during data synchronization.

[b] This conflict can be resolved by log replay. No update or file will be lost in data synchronization. Old data versions are always captured in logs for later retrieval.

[c] In data synchronization, the copy of p at fs_2 is propagated to fs_1 for a delete/update conflict or a delete/create conflict. In the case of a delete/rename conflict, p is simply omitted in both systems and the newly introduced filename is used to create a file at fs_1. The "file miss "problem remains unsolved.

[d] This conflict will occur if the file p was deleted at fs_1 while a new file was created with the name p after the original file with the name p had been removed from fs_2. Thus, before synchronization, p refers to the newly created file in fs_2, instead of to the original one.

[e] In practice, this problem is treated as an update/update conflict and can be solved regardless of identities. The "identity loss" problem will remain unsolved (even undetected).

[f] Such conflicts can be detected and resolving them may involve name clashes.

[g] This conflict will occur if the file p was updated at fs_1 while a new file was created with the name p after the original file with the name p had been removed from fs_2. Thus, before synchronization, p refers to the newly created file in fs_2, instead of to the original one.

* A conflict that is discovered in this analysis for the first time.

related conflicts often gives rise to name clashes. Structure conflicts can be detected and (manually) resolved.

Both pathname-based synchronization and identification-based synchronization are based on the "current" states of file systems. Log-based synchronization is based on historical information of data access.

Log-based synchronization consists of rolling back system states, serializing disconnected updates, and replaying serialized disconnected updates. This method can handle updating, renaming, deletion and creation. Notably, this method can detect deletions and thus produces better solutions to deal with file miss in deletion-related conflicts. Still, name clashes may occur in resolving rename-related conflicts. In this case, manual repair is often needed. In handling structures conflicts, the effectiveness of log-based synchronization relates to how files are referenced in logs. When names are used, name-link conflicts can neither be solved nor detected and node-typing conflicts can be detected and solved in ad hoc ways. When identifications are used, both name-link conflicts and node-typing conflicts can be solved.

2.5 Identity-based history synchronization

In data synchronization, the identification-based approach avoids identity loss. It provides more meaningful results than the pathname-based approach does. The identification-based approach has been implemented in many systems, as discussed in Section 1.6. In those systems, associations between identifications of copies of data objects in different devices must be retained. In Section 2.4.4 the data structure *IMAP* was designed for this purpose.

In home environments, any two devices involved in data transfer may belong to different systems. Maintaining the associations between identifications of data copies that are stored in different systems is difficult to achieve in home environments for several reasons. In the first place, as explained in Section 1.2, devices in home environments are manufactured by a variety of vendors and an enormous amount of effort is needed to achieve device interoperability. Secondly, association tables are distributed over many devices and maintaining the consistency of those tables in dynamic home environments is difficult. Thirdly, maintaining association tables consumes system resources, such as memory, storage and CPU time. For miniaturized portable consumer electronic devices this is a burden. Therefore, more practical solutions are needed so that identification associations can be maintained at a minimal cost.

Log-based synchronization can be used to handle delete-related conflicts. Usually logs are treated as system data. In data migration, logged information related to the data being shipped is not transferred. After migration, data objects lose their histories. Moreover, due to technical restrictions explained in Section 2.4.6, a properly designed log mechanism is required to assist the synchronization at a minimal cost.

Identities. Preserving identities of data objects is a technically practical solution, instead of maintaining associations, in heterogeneous home environments. Each digitized data object, for example a digital photo or a working document, has a logical identity in the user's mind. A data object obtains its immutable identity on its creation and the identity is preserved in data migration, no matter on which device the data object is stored or used. Providing a query on the identity of a given data object requires little overhead on the devices.

The name of a data object is treated as one aspect of the data object, like the byte data of the object. Modifying any aspect of an object does not change the identity of the object at all. Renaming a data object will have the identity of the object untouched. Names can be used to look up data objects. However, one name may correspond to multiple data objects.[8] To reliably access unique data, logical identities should be used.

In implementation, logical identities of data objects can be captured by globally unique identifiers. In an HDMS, a globally unique identifier is assigned to a data object, no matter on which device the data object is created. The identifier is permanently attached to the data object and is persistent and immutable. When the data object migrates between content directories, the identifier of the object will remain untouched. The logical identity of the data object will be shipped together with other metadata of the object.

Sacrificing association tables makes it an expensive operation to locate all copies of a data object. This drawback is not remarkable, however, since the eventual consistency requirement defined in Section 1.3 is achieved by the iteration of data synchronization between any two devices. Each round of data synchronization is invoked, for example, when two devices are reconnected or when the user intends to do so. Moreover, providing identity query requires less overhead on the devices than maintaining association tables.

Identity-based history synchronization. While data identities are maintained, it could still happen that a data object is copied to several devices. Different copies have the same logical identity in the user's perspective. An *access history* is associated with each data copy to record disconnected updates on the copy. Inconsistencies between copies can be resolved by using the identity information and the recorded information in logs.

How different types of inconsistencies are handled in identity-based synchronization is summarized in Table 2.3. In the first place, the create operation no longer interferes with the rename operation, since newly created data objects have new identities. Update conflicts and deletion conflicts can be handled using log replay.

Secondly, a name is treated as a property of a data object, so renaming a data object is regarded as an update on the object. In dealing with an update/rename conflict, the update operation and the rename operation are modifications of two different properties of the same data object and they do not really conflict. Commuting the operations can solve the problem. A rename/rename conflict can be treated as an update/update conflict, since both operations try to modify the same aspect of the data.

[8] An HDMS is an aggregation of several content directories on a home network. When a pathname is used to retrieve data from an HDMS system, the pathname is used to retrieve data objects from individual content directories. As a result, a collection of objects will be retrieved, although in each content directory a pathname maps to at most one data object at a time. Applications developed on top of an HDMS should be adapted to this change. In photo-browsing-like applications in home environments, user interfaces could handle such situations.

Table 2.3. Disconnected updates in identity-based history synchronization. Assume two file systems, fs_1 and fs_2, and a pathname p. The "Disconnected updates" column lists all combinations of operations that can be applied to fs_1 and fs_2 individually which may introduce inconsistencies in other data synchronization approaches. The "Identity-based history synchronization" column illustrates how those problems are solved in identity-based synchronization.

Problematic disconnected updates		Identity-based history synchronization	
		Applicability	Solvability
update	update/update	update/update	solved(log replay)[a]
deletion	delete/update	delete/update	solved(log replay)
	delete/create	-[b]	
	delete/rename	delete/update	solved(log replay)
naming	update/rename	-	
	update/create	-	
	create/create	-	
	create/rename	-	
	rename/rename	update/update[c]	solved(log replay)
structure	name-link	name-link	solved
	node-typing	-[d]	

[a] This conflict can be resolved by log replay.
[b] Not applicable. The newly created data object has a different identity from the object that was updated. Such a conflict is avoided in data synchronization.
[c] The name becomes one aspect (property) of the data object. A rename operation is an "update" on the name property of the data object. The rename/rename conflict can be treated as an update/update conflict.
[d] Files and directories always have different identities. Name clashes might occur.

Thirdly, a name-link conflict can also be resolved. In the example illustrated in Diagram A of Figure 2.5, a new "*Favorites\flowers*" file is created using the data of the file having the same name on the server at home while a new link to the "*Favorites\img*1344*.jpg*" file is created using "*Favorites\flowers*" on the file server at home. In this case name clashes occur and can be reported to the user.

Related work. The requirement of preserving data identities among heterogeneous systems and devices in home environments is being adopted both in academia and in industries. In Ficus, Coda or Rover, logical identities are used mainly within individual systems, not for inter-system data exchange.

Logical content [149] is used in video content management, where files of different formats are used to store the same content. Logical content makes it possible to manage a system that uses files of different formats to store the same content to be used by different modules [149]. It also helps to manage video content consisting of multiple physical files.

On the Internet, data objects are identified by Uniform Resource Names (URNs) [102, 139] and Uniform Resource Identifiers (URIs) [14, 15]. COM components [96] have Class Identifiers, which can be translated into Global

Unique identifiers (GUIDs). The GUID can be used by the installer as a valid product code, package code, or component code. Each Jini [143] service is assigned a universally unique identifier (UUID) on its registration. Applications can use UUIDs to identify and access persistent services. In the SQL Server [97], GUIDs can be assigned to records in database tables so that records can be migrated to different systems. The GUID.org [137] web service assigns anonymous user IDs to web browsers so that web sites can recognize users when they return.

2.6 Concluding remarks

In this chapter, file systems were used as research carriers in analyzing disconnected updates. Various conflicts were categorized into four types: update conflicts, deletion conflicts, naming conflicts, and structure conflicts. Incautious conflict resolution can lead to update loss, file miss, identity loss, or name clash. In the analysis, the Z notation was used in defining various operations and disconnected updates. This formal approach yielded a good understanding of disconnected updates.

Regarding conflict resolution, three data synchronization methods were discussed: pathname-based synchronization, ID-based synchronization, and log-based synchronization. Despite the fact that user knowledge and user involvement can help to obtain decent results in conflict resolution, it was described how each method performs in resolving different types of conflicts.

- Pathnames are not reliable for identifying data objects. Pathname-base synchronization does not address the identity loss problem. Nor does it tackle structure conflicts. However, it does not introduce name clashes.
- Identification helps to address the identity loss problem. Identification-based synchronization handles update/update conflicts and deletion-related conflicts in the same way as pathname-based synchronization. However, name clashes can occur in resolving rename-related conflicts and structure conflicts should be manually repaired.
- Log-based synchronization can handle updating, renaming, deletion and creation. Still, name clashes may occur in resolving rename-related conflicts. In this case, manual repair is often needed.
- Compared with pathname-based synchronization and identification-based synchronization, which are state-based, log-based synchronization consumes more storage, memory and network bandwidth resources in storing and commuting logs, which may slow down portable devices with limited system and network resources.

As a synthesis of the best of all three approaches, the identity-based history synchronization was proposed in this chapter for dealing with disconnected updates in home environments. In this approach, all data objects are globally uniquely identified. Identities are immutable and persistent. They are shipped together with the data objects during data migration. In this way the identity loss problem can be solved. Names of data objects are treated as properties of data objects. Renaming a data object is recorded in a log of the data object. Conflicting renaming operations are simply performed in a time order. It is always possible to inspect all performed operations in logs. In identity-based history synchronization, conflicting disconnected updates are update/update conflicts and update/delete conflicts.

Part II

Formal development of data synchronization

3. Formal model of disconnected updates

In this chapter, a formal model of an in-home data management system (HDMS) will be described. Normal logs that can be used in an HDMS to record disconnected updates will be modelled. Validity requirements relating to entries of logs will be investigated. The models will finally be synthesized to arrive at a full formal specification of an HDMS.

3.1 Overview of the model

An HDMS supporting disconnected updates, called a DU system, consists of a collection of *data spaces*. Data spaces represent storage devices. For example, a DU system $\mathbf{S}_{\mathrm{DU}}$ has three component data spaces.

$$\mathbf{S}_{\mathrm{DU}} = \{S_1, S_2, S_3\}.$$

A DU system contains a collection of *data items*. Each data item has a name part and a value part, called *key* and *value*, respectively. A data item is uniquely identified by its key in the system. The key of a data item is immutable while its value can be modified. For example, the system $\mathbf{S}_{\mathrm{DU}}$ has three data items:

$$(x, 1), (y, 2), (z, 3),$$

where x, y and z denote keys of data items. For simplicity, the values of data items will be assumed to be natural numbers here. In a real system, a data item may refer to a digital photo, for example. The name part of the data item then models the photo's identity and the value part of the data item models the photo's byte information.

A data item may have several copies in a DU system. Copies of a data item have the same values in their name parts and may have different values in their value parts. For example, in the DU system $\mathbf{S}_{\mathrm{DU}}$, copies of x have the same keys in S_1, S_2 and S_3.

$$\begin{aligned}
S_1 &= \{(x, 1), (y, 2), (z, 3)\}\\
S_2 &= \{(x, 1), (y, 2)\}\\
S_3 &= \{(x, 1), (u, 4), (v, 5)\}
\end{aligned}$$

Formally, the states of a DU system can be defined as mappings from data spaces and keys of data items to values of data items, as follows.

Definition 3.1.1. *The state of a DU system is a triple* $(S, K, V, Store)$*.* S *is a set of data spaces.* K *is a set of keys.* V *is a set of values.* $Store : S \times K \nrightarrow V$ *is a partial function which describes where current values of data items are stored.*

3.1.1 Consistencies

In the scenario analysis in Chapter 2, four operations were identified: read, update, create and delete. The four operations are modelled by *read*, *write*, *add* and *delete*, respectively, in a DU system. These operations can be performed one at a time on any data item stored in a data space. They are assumed to be atomic.

Definition 3.1.2. *In a DU system, the following operations are allowed.*

- *read, read the value of a data item from a data space.*
- *write, update the value of a data item in a data space.*
- *add, add a data item to a data space.*
- *delete, remove a data item from a data space.*

In a DU system, a copy of a data item can be introduced by using the *add* operation. There will then be no difference between the original data item and the newly created copy. The original data item and the newly created copy are treated in the same way in the system. In a DU system, any modification on a data item is performed one data space at a time. One copy of a data item can be updated using the *write* operation, independently of the other copies of the same data item in a DU system.

Definition 3.1.3 (Disconnected update). *An operation is a* disconnected update *if it changes the current value of a data item in a data space, independently of the other copies of the data item in other data spaces.*

Due to disconnected updates, a data item may have different values in data spaces. For example, consider a system state of a DU system $\mathbf{T}_{\mathrm{DU}}$, in which copies of x have different values in S_1, S_2 and S_3.

$$S_1 = \{(x,1),(y,2),(z,3)\}$$
$$S_2 = \{(x,4),(y,2)\}$$
$$S_3 = \{(x,5),(u,4),(v,5)\}$$

Definition 3.1.4 (Data consistency). *A data item is* consistent *in a system state of a DU system if it has the same value as all the other copies of it in the state.*

According to this definition, a data item with the key x is consistent in a DU system if all the data items with the key x in the system have the same value in their value part; otherwise, the data item is inconsistent (not consistent). In a practical example, this definition expresses the requirement that different image files with the same identity (from the same digital photo) should have the same image content. In the above-mentioned examples, x is consistent in $\mathbf{S}_{\mathrm{DU}}$ and is inconsistent in $\mathbf{T}_{\mathrm{DU}}$.

Definition 3.1.5 (System consistency). *A system state of a DU system is* consistent *if each data item is consistent in the state.*

According to this definition, for each key x, all data items in the system with the key x should have the same value in the value part. In the above-mentioned examples, $\mathbf{S}_{\mathrm{DU}}$ is consistent while $\mathbf{T}_{\mathrm{DU}}$ is inconsistent.

To resolve data inconsistencies and system inconsistencies, the operation *sync* is introduced.[1]

Definition 3.1.6 (Synchronization). sync *is an operation of a DU system with the effect that a data item and a copy of the data item will have the same value after the execution of the operation, thus making the system consistent.*

3.1.2 Logs

In a DU system, data consistency and system consistency can be easily verified. Resolving inconsistencies, however, is not an easy task, which will often be fairly application-dependent or user-dependent. Resolving inconsistencies in a sensible way requires additional information. Historical information on how inconsistencies are introduced can be of great importance in conflict resolution. In real systems, log files are used to capture such information.

In a DU system, *logs* are used to record data accesses on data spaces. Generally speaking, a log is a sequence of log entries. A *log entry* contains information on the time, the operation and the accessed data item of an access. A log entry also keeps record of the values of the accessed data item before and after the access. For example, suppose the log of the data space S_1 is l_1.

$$l_1 = \langle A(x), W(x)1, R(x)1, A(y), W(y)2, A(z), W(z)3, D(x)\rangle,$$

where A, W, R and D stand for add, write, read and delete, respectively;[2] x is the key of the accessed data item; timestamps are omitted. $A(x)$ reads "add x to the data space". $W(x)1$ reads "write the value 1 to x". $R(x)1$ reads "read the value of x and the value is 1". $D(x)$ reads "delete x from the

[1] *sync* is an abbreviation of "synchronize".

[2] Without confusion, A and add are used interchangeably. The same holds for W and write, R and read, D and delete.

data space". The values of the accessed data item before and after the access can be derived from the context.[3]

Log size, i.e. the number of log entries of a log, may be limited or unlimited. If it is unlimited, each access gives rise to a new log entry, which is appended at the end of the log; if it is limited, a log might overflow.

3.1.3 Structure of a formal model

In the subsequent sections, a full formal specification of a DU system will be built step-by-step using the Z notation. First, data spaces and operations on data spaces will be modelled. Next, logs will be formalized, after which DU systems and operations on DU systems will be built up in a structured fashion. Finally, the *sync* operation will be formally specified.

3.2 Data spaces

A data space consists of a number of data items. A data item has a key and a value. Data items are identified by their names. The key of a data item is unique in the data space to which the item belongs. For the purpose of this specification, two basic types are introduced. *KEY* denotes the set of keys of data items. *VAL* denotes the set of values of data items.

$$[KEY, VAL]$$

A data space is modelled as a partial function, since a key should not be associated with two different data items within a data space. The *DataSpace* schema specifies the current state of a data space.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{DataSpace}} \\
\hline
\textit{content} : KEY \pfun VAL \\
\hline
\end{array}$$

Initially, a data space has no content. The *DataSpaceInit* schema models a data space in which the *content* function is empty.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{DataSpaceInit}} \\
\hline
\textit{DataSpace} \\
\hline
\operatorname{dom} \textit{content} = \emptyset \\
\hline
\end{array}$$

The following two schemas may be used to specify operations on data spaces.

[3] This notation was introduced by Ahamad et al. in a study of processor consistency [5].

$$\begin{array}{l} \Delta DataSpace \mathrel{\widehat=} [DataSpace;\ DataSpace'] \\ \Xi DataSpace \mathrel{\widehat=} [\Delta DataSpace \mid \theta DataSpace = \theta DataSpace'] \end{array}$$

$\Delta DataSpace$ defines two variables, $DataSpace$ and $DataSpace'$, denoting the state of a data space before and after a state change, respectively. $\Xi DataSpace$ requires that both variables have identical bindings. If an operation changes the state of a data space, $\Delta DataSpace$ is included in the specification of the operation; otherwise, $\Xi DataSpace$ is included.

A data space can be accessed by add, read, write and delete operations. For an add operation to succeed, the argument key must not be in use.

$$\begin{array}{|l} \multicolumn{1}{l}{DAdd_0} \\ \hline \Delta DataSpace \\ k? : KEY \\ v? : VAL \\ \hline k? \notin \operatorname{dom} content \\ content' = content \cup \{k? \mapsto v?\} \\ \hline \end{array}$$

For a read operation to succeed, the argument key must be in use. If this is indeed the case, the value of the denoted data item will be returned.

$$\begin{array}{|l} \multicolumn{1}{l}{DRead_0} \\ \hline \Xi DataSpace \\ k? : KEY \\ v! : VAL \\ \hline k? \in \operatorname{dom} content \\ v! = content(k?) \\ \hline \end{array}$$

A successful write operation will replace the old value of a data item. The argument key must be in use.

$$\begin{array}{|l} \multicolumn{1}{l}{DWrite_0} \\ \hline \Delta DataSpace \\ k? : KEY \\ v? : VAL \\ \hline k? \in \operatorname{dom} content \\ content' = content \oplus \{k? \mapsto v?\} \\ \hline \end{array}$$

A successful delete operation will remove a data item from the data space. As with the read and write operations, the argument key must be in use.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{DDelete}_0} \\ \hline
\Delta \textit{DataSpace} \\
k? : \textit{KEY} \\ \hline
k? \in \operatorname{dom} \textit{content} \\
\textit{content}' = \{k?\} \lhd \textit{content} \\ \hline
\end{array}$$

To complete the specification , the *REPORT* type is added to the formal specification. *REPORT* defines a set of values that indicate various return codes (statues of operations).

$$\textit{REPORT} ::= \textit{key\_in\_use} \mid \textit{key\_not\_in\_use} \mid \textit{success}$$

success indicates a successful operation. *key_in_use* indicates an error in an operation, saying that the supplied key is being used. *key_not_in_use* indicates an error due to the supplied key not being in use.

The output variable, *result*!, is added to each operation, indicating the status of the operation. The *Success* schema specifies a successful operation.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Success}} \\ \hline
\textit{result}! : \textit{REPORT} \\ \hline
\textit{result}! = \textit{success} \\ \hline
\end{array}$$

The *KeyInUse* schema specifies an unsuccessful operation, where the supplied key is in use.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{KeyInUse}} \\ \hline
\Xi \textit{DataSpace} \\
k? : \textit{KEY} \\
\textit{result}! : \textit{REPORT} \\ \hline
k? \in \operatorname{dom} \textit{content} \\
\textit{result}! = \textit{key\_in\_use} \\ \hline
\end{array}$$

The *KeyNotInUse* schema specifies an unsuccessful operation, where the supplied key is not in use.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{KeyNotInUse}} \\ \hline
\Xi \textit{DataSpace} \\
k? : \textit{KEY} \\
\textit{result}! : \textit{REPORT} \\ \hline
k? \notin \operatorname{dom} \textit{content} \\
\textit{result}! = \textit{key\_not\_in\_use} \\ \hline
\end{array}$$

The full specifications of operations on data spaces are as follows.

$$\begin{aligned}
&\mathit{DAdd} && \mathrel{\widehat=} (\mathit{DAdd}_0 \land \mathit{Success}) \lor \mathit{KeyInUse}\\
&\mathit{DRead} && \mathrel{\widehat=} (\mathit{DRead}_0 \land \mathit{Success}) \lor \mathit{KeyNotInUse}\\
&\mathit{DWrite} && \mathrel{\widehat=} (\mathit{DWrite}_0 \land \mathit{Success}) \lor \mathit{KeyNotInUse}\\
&\mathit{DDelete} && \mathrel{\widehat=} (\mathit{DDelete}_0 \land \mathit{Success}) \lor \mathit{KeyNotInUse}
\end{aligned}$$

Each of those four operations is *total*, meaning that the precondition of the operation is *true*. The operations are always defined, given any valid data space state.

3.3 Normal logs

Logs consist of log entries. A log entry records an execution of an access on a data object. It contains the execution time, the performed operation, the data object on which the access was performed, the value of the data object before the execution, and the value of the data object after the execution.

First, some basic type definitions will be introduced, which will be used throughout this chapter. *OP* defines a set of values that indicate the type of an access.

$$\mathit{OP} ::= \mathit{add} \mid \mathit{read} \mid \mathit{write} \mid \mathit{delete}.$$

add indicates an add operation. *read* indicates a read operation. *write* indicates a write operation. *delete* indicates a delete operation.

The type *TIME* is also introduced. The elements of *TIME* are used in logging when an access was executed. For simplicity, assume the elements of *TIME* to be natural numbers, as in more practical systems.

$$\mathit{TIME} == \mathbb{N}$$

Log entries can be formally defined as a quintuple of the access time, the operation type, the accessed data item, the pre-value and post-value of the data item.

$$\mathit{LogEntry} == (((\mathit{TIME} \times \mathit{OP}) \times \mathit{KEY}) \times \mathit{VAL}) \times \mathit{VAL}$$

All projection functions relating to accessing components of a log entry are defined as follows.

$$\begin{array}{|l}
\mathit{time} : \mathit{LogEntry} \to \mathit{TIME}\\
\hline
\forall\, e : \mathit{LogEntry} \bullet \mathit{time}(e) = \mathit{first}(\mathit{first}(\mathit{first}(\mathit{first}(e))))
\end{array}$$

$$\begin{array}{|l}
\mathit{op} : \mathit{LogEntry} \to \mathit{OP}\\
\hline
\forall\, e : \mathit{LogEntry} \bullet \mathit{op}(e) = \mathit{second}(\mathit{first}(\mathit{first}(\mathit{first}(e))))
\end{array}$$

$$\begin{array}{|l} key : LogEntry \rightarrow KEY \\ \hline \forall\, e : LogEntry \bullet key(e) = second(first(first(e))) \end{array}$$

$$\begin{array}{|l} prev : LogEntry \rightarrow VAL \\ \hline \forall\, e : LogEntry \bullet prev(e) = second(first(e)) \end{array}$$

$$\begin{array}{|l} post : LogEntry \rightarrow VAL \\ \hline \forall\, e : LogEntry \bullet post(e) = second(e) \end{array}$$

Logs can be modelled as sequences of log entries that are ordered by time.[4] It is due to the fact that operations that write data into a log file are usually sequentially ordered in a local file system. Thus, it is here assumed that accesses on a data space are sequential. Formally, logs are defined as follows.

$$Log == \{l : \operatorname{seq} LogEntry \mid isOrdered(l)\}.$$

The predicate *isOrdered* checks whether the entries of a sequence of log entries are ordered by time.[5]

$$\begin{array}{|l} isOrdered\_ : \mathbb{P}(\operatorname{seq} LogEntry) \\ \hline \forall\, l : \operatorname{seq} LogEntry \bullet isOrdered(l) \Leftrightarrow \\ \qquad \forall\, i, j : \operatorname{dom} l \bullet i < j \Rightarrow time(l(i)) < time(l(j)) \end{array}$$

The objects defined by *Log* are named *log instances.* Without confusion, logs and log instances will be used interchangeably in the rest of the chapter. The

[4] In this model, log entries can be ordered by their timestamps and they also preserve the time ordering in a log. This redundancy is introduced largely because this model is close to the data structure in implementation. Moreover, the most recent operation is often used in data synchronization, as will be explained in Chapter 5. This information can be easily retrieved from a log in this model, since the last element of a log is the most recent access. Alternatively, a log can be modelled as a partial mapping l,

$$l : TIME \rightarrow\!\!\!\!\!+ ((OP \times KEY) \times VAL) \times VAL.$$

Since elements of *TIME* are totally ordered, the ordering of accesses can still be retrieved from the mapping. The alternative model does not reveal details of implementation.

[5] The predicate is defined in terms of a set of objects that satisfy it (see pages 81–82 in [174]). It is convenient to treat the name of the set as a unary operator. In this case, the definition includes an underscore to indicate the position of the argument. Usually, it will not be necessary to put brackets around the argument when parameterizing the predicate. For ease of reading, brackets will be added whenever they improve the readability.

predicate *isOrdered* requires that there are no two entries in a log having the same timestamp. Note that the λ-notation has been used in the definition of *isOrdered* for simplicity.[6] An access is usually recorded by appending a log entry to the end of a log file. In practice, it is easy to guarantee that any two appending operations on the same log file will always be sequentially executed.

One basic operation on a log is the *append* operation. According to the assumption that accesses on a data space are sequentially ordered, one timestamp must never appear twice in a log. When appending an entry to a log it should moreover be ensured that the timestamp of the log entry comes after that of the last log entry of the log, so that the log entries of the resulting log will still preserve time order. The uniqueness and order-preserving requirements are captured by the predicate *isAfterLast*.

$$\begin{array}{|l}
\mathit{isAfterLast}\_ : \mathbb{P}(\mathrm{seq}\,\mathit{LogEntry} \times \mathit{TIME}) \\
\hline
\forall\, l : \mathrm{seq}\,\mathit{LogEntry};\ t : \mathit{TIME} \bullet \mathit{isAfterLast}(l,t) \Leftrightarrow \\
\quad \textbf{if } l = \langle\rangle \textbf{ then } \mathit{true} \textbf{ else } \mathit{time}(\mathit{last}(l)) < t
\end{array}$$

Lemma 3.3.1. *Given $l : Log$ and $t : TIME$,*

$$\mathit{isAfterLast}(l,t) \equiv \mathit{isAfter}(l,t)$$

where

$$\begin{array}{|l}
\mathit{isAfter}\_ : \mathbb{P}(\mathrm{seq}\,\mathit{LogEntry} \times \mathit{TIME}) \\
\hline
\forall\, l : \mathrm{seq}\,\mathit{LogEntry};\ t : \mathit{TIME} \bullet \mathit{isAfter}(l,t) \Leftrightarrow \\
\quad \forall\, i : \mathrm{dom}\, l \bullet \mathit{time}(l(i)) < t
\end{array}$$

Proof. The proof is divided into two cases.[7] **(1)** If $l = \langle\rangle$, trivial. **(2)** If $l \neq \langle\rangle$,

$$\begin{array}{ll}
& \mathit{isAfterLast}(l,t) \wedge \{l \text{ is a log}\} \\
\equiv & \{\text{Definition of } \mathit{isAfterLast}\} \\
& \mathit{time}(\mathit{last}(l)) < t \wedge \{l \text{ is a log}\} \\
\equiv & \{l \text{ is a log, thus } \mathit{isOrdered}(l) \text{ holds.}\} \\
& \mathit{time}(\mathit{last}(l)) < t \wedge (\forall\, i,j : \mathrm{dom}\, l \bullet i < j \Rightarrow \mathit{time}(l(i)) < \mathit{time}(l(j))) \\
\equiv & \{\text{Let } j = \#(l). \text{ Conjunct with the fact that } l \text{ is a log.}\}
\end{array}$$

[6] In the **Z** notation, λ-notation is a concise alternative to specify sets of ordered n-tuples [174, 36].

$$\lambda\, x_1 : X_1;\ ...;\ x_n : X_n \mid P \bullet t = \{x_1 : X_1;\ ...;\ x_n : X_n \mid P \bullet (x_1, ..., x_n) \mapsto t\}.$$

The constraint part P of a lambda expression is often omitted in this thesis.

[7] In the proof, the proof format proposed by W.H.J. Feijen is adopted, as used in [35]. In the format, two consecutive proof stages are separated by a connective ($\equiv$, $\Leftarrow$, or $\Rightarrow$) and a justification.

$$\begin{aligned}
&\quad \mathit{time}(\mathit{last}(l)) < t \land (\forall\, i : \mathrm{dom}\, l \bullet i < \#(l) \Rightarrow \\
&\qquad \mathit{time}(l(i)) < \mathit{time}(l(\#(l)))) \land \{l \text{ is a log}\} \\
&\equiv \{l(\#(l)) = \mathit{last}(l),\ \mathit{time}(\mathit{last}(l)) < t\} \\
&\quad \mathit{time}(l(\#(l))) < t \land (\forall\, i : \mathrm{dom}\, l \bullet i < \#(l) \Rightarrow \mathit{time}(l(i)) < t) \land \\
&\qquad \{l \text{ is a log}\} \\
&\equiv \{\text{Predicate logic}\} \\
&\quad (\forall\, i : \mathrm{dom}\, l \bullet (i < \#(l) \Rightarrow \mathit{time}(l(i)) < t) \land \\
&\qquad (i = \#(l) \Rightarrow \mathit{time}(l(i)) < t)) \land \{l \text{ is a log}\} \\
&\equiv \{\text{Predicate logic}\} \\
&\quad (\forall\, i : \mathrm{dom}\, l \bullet \mathit{time}(l(i)) < t) \land \{l \text{ is a log}\} \\
&\equiv \{\text{Definition of } \mathit{isAfter}\} \\
&\quad \mathit{isAfter}(l, t) \land \{l \text{ is a log}\}
\end{aligned}$$

□

The *append* function is defined as follows.

$$\begin{array}{|l}
\mathit{append} : \mathit{Log} \times \mathit{LogEntry} \rightarrow \mathrm{seq}\, \mathit{LogEntry} \\
\hline
\mathit{append} = \lambda\, l : \mathit{Log};\ e : \mathit{LogEntry} \bullet \\
\qquad \mathbf{if}\ \mathit{isAfterLast}(l, \mathit{time}(e))\ \mathbf{then}\ l \frown \langle e \rangle\ \mathbf{else}\ l
\end{array}$$

The *append* function is a type-safe function in the sense that applying it to a log and a log entry still results in a log, which is stated by the following theorem.

Theorem 3.3.1. *Given $l : \mathit{Log}$ and $e : \mathit{LogEntry}$, $\mathit{append}(l, e)$ is still a log.*

Proof. To prove that $\mathit{append}(l, e)$ is still a log, it must be shown that

$$\mathit{isOrdered}(\mathit{append}(l, e))$$

holds. The proof is divided into three cases. **(1)** If $l = \langle\rangle$, trivial. **(2)** If $l \neq \langle\rangle$ and $\mathit{isAfterLast}(l, \mathit{time}(e)) = \mathit{false}$, trivial. **(3)** If $l \neq \langle\rangle$ and $\mathit{isAfterLast}(l, \mathit{time}(e)) = \mathit{true}$,

$$\begin{aligned}
&\quad \mathit{isOrdered}(\mathit{append}(l, e)) \\
&\equiv \{\text{Definition of } \mathit{append}\} \\
&\quad \mathit{isOrdered}(l \frown \langle e \rangle) \\
&\equiv \{\text{Definition of } \mathit{isOrdered}\} \\
&\quad \forall\, i, j : \mathrm{dom}\, l \frown \langle e \rangle \bullet \\
&\qquad i < j \Rightarrow \mathit{time}(l \frown \langle e \rangle(i)) < \mathit{time}(l \frown \langle e \rangle(j))
\end{aligned}$$

Given any i and j from $\mathrm{dom}\, l \frown \langle e \rangle$ and $i < j$, it needs to be proven that

$$\mathit{time}(l \frown \langle e\rangle(i)) < \mathit{time}(l \frown \langle e\rangle(j)) \tag{3.1}$$

If $j < \#(l \frown \langle \mathit{time}(e)\rangle) = \#(l) + 1$, then $i < j \leq \#(l)$.

$$\begin{aligned}
&(3.1)\\
\equiv\ &\{i < j \leq \#(l)\}\\
&\mathit{time}(l(i)) < \mathit{time}(l(j))\\
\Leftarrow\ &\{\text{Definition of } \mathit{Log},\ \mathit{isOrdered}(l) \text{ holds}\}\\
&l \text{ is a log}
\end{aligned}$$

If $j = \#(l \frown \langle e\rangle)$, then $\mathit{time}(l \frown \langle e\rangle(j)) = \mathit{time}(e)$. By the assumption $\mathit{isAfterLast}(l, \mathit{time}(e)) = \mathit{true}$, $\mathit{time}(\mathit{last}(l)) < \mathit{time}(e)$ holds. Thus, if $i < \#(l)$, then

$$\begin{aligned}
&(3.1)\\
\equiv\ &\{i < \#(l)\}\\
&\mathit{time}(l(i)) < \mathit{time}(l \frown \langle e\rangle(j))\\
\equiv\ &\{j = \#(l \frown \langle e\rangle)\}\\
&\mathit{time}(l(i)) < \mathit{time}(e)\\
\Leftarrow\ &\{\mathit{time}(\mathit{last}(l)) < \mathit{time}(e), \text{ transition of } <\}\\
&\mathit{time}(l(i)) < \mathit{time}(\mathit{last}(l))\\
\Leftarrow\ &\{\text{Definition of } \mathit{Log},\ \mathit{isOrdered}(l) \text{ holds}\}\\
&l \text{ is a log}
\end{aligned}$$

If $i = \#(l)$, then

$$\begin{aligned}
&(3.1)\\
\equiv\ &\{i = \#(l),\ \mathit{last}(l) = l(i) = l \frown \langle e\rangle(i)\}\\
&\mathit{time}(\mathit{last}(l)) < \mathit{time}(l \frown \langle e\rangle(j))\\
\equiv\ &\{\mathit{time}(l \frown \langle e\rangle(j)) = \mathit{time}(e),\ \mathit{time}(\mathit{last}(l)) < \mathit{time}(e)\}\\
&\mathit{true}
\end{aligned}$$

□

Given a log $l : \mathit{Log}$ and a $k : \mathit{KEY}$, the *filter* function calculates all log entries that are related to k.

```
$\mathit{filter} : \mathrm{seq}\,\mathit{LogEntry} \times \mathit{KEY} \to \mathrm{seq}\,\mathit{LogEntry}$
────────────
$\mathit{filter} = \lambda\, l : \mathrm{seq}\,\mathit{LogEntry};\ k : \mathit{KEY} \bullet$
    if $l = \langle\rangle$ then $\langle\rangle$
    else if $\mathit{key}(\mathit{head}(l)) = k$ then $\langle \mathit{head}(l)\rangle \frown \mathit{filter}(\mathit{tail}(l), k)$
        else $\mathit{filter}(\mathit{tail}(l), k)$
```

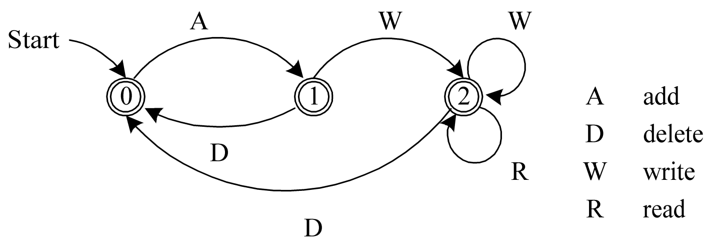


Figure 3.1. Valid behaviors of the operations that can be performed on a given key.

The *operations* function retrieves operations from log entries.

$$
\begin{array}{|l}
\mathit{operations} : \mathrm{seq}\, \mathit{LogEntry} \rightarrow \mathrm{seq}\, \mathit{OP} \\
\hline
\mathit{operations} = \lambda\, l : \mathrm{seq}\, \mathit{LogEntry} \bullet \\
\quad \mathbf{if}\ l = \langle\rangle\ \mathbf{then}\ \langle\rangle\ \mathbf{else}\ \langle \mathit{op}(\mathit{head}(l)) \rangle \frown \mathit{operations}(\mathit{tail}(l))
\end{array}
$$

Given a log and a key, the *ops* function retrieves all operations of log entries that are related to the given key.

$$
\begin{array}{|l}
\mathit{ops} : \mathrm{seq}\, \mathit{LogEntry} \times \mathit{KEY} \rightarrow \mathrm{seq}\, \mathit{LogEntry} \\
\hline
\mathit{ops} = \lambda\, l : \mathrm{seq}\, \mathit{LogEntry};\ k : \mathit{KEY} \bullet \mathit{operations}(\mathit{filter}(l, k))
\end{array}
$$

3.3.1 Valid logs

A successful execution of an operation on a data space has certain constraint requirements, as specified in Section 3.2. For example, adding a data item to a data space will succeed only if the data space does not yet contain the key of the data item. In addition, given a key, the read, write, and delete operations can only be performed after the add operation. In other words, a key should be declared before being used. Such constraints on executions of operations result in the fact that only a subset of objects defined by the type Log occur in the model. The logs that record successful executions of operations on data spaces are called *valid* logs. In this section, the validity of logs will be formally specified.

Figure 3.1 illustrates the constraint requirements of executing an operation on a key of a data space by using a finite-state machine.

Definition 3.3.1. *The finite-state machine FSM_c expressing the constraint requirements regarding the operations on a key is defined as follows.*

1. *The input alphabet Σ is $\{A, W, R, D\}$.*
2. *The set of states S is $\{0, 1, 2\}$.*
3. *The initial state is 0.*

4. *The accepting states are* $0, 1, 2$.
5. *The transition function is a partial function,* $move : S \times \Sigma \nrightarrow S$. *It is defined as follows:* $move(0, A) = 1$, $move(1, D) = 0$, $move(1, W) = 2$, $move(2, W) = 2$, $move(2, R) = 2$ *and* $move(2, D) = 0$.

For example, $AWRD$ can be accepted by FSM_c, while WR and AWA can not. The set of all the sentences that can be accepted by FSM_c is denoted by $\mathcal{L}(FSM_c)$. Given a sentence $s \in \mathcal{L}(FSM_c)$, the *accepting state* of s is obtained by successive application of the elements of s as follows, $move(...move(move(0, s(1)), s(2))..., s(\#s))$.

Lemma 3.3.2. *Given any nonempty sentence* $s \in \mathcal{L}(FSM_c)$*, its accepting state is determined by its last element.*

Proof. According to Definition 3.3.1, any two valid moves will have the same resulting state if they have the same input, as shown as follows

- $move(0, A) = 1$.
- $move(1, D) = 0$, $move(2, D) = 0$.
- $move(1, W) = 2$, $move(2, W) = 2$.
- $move(2, R) = 2$.

□

The finite-state machine approach defines validity constraint requirements by modelling dynamic behaviors. This goal can also be achieved by directly using predicates. The constraint requirements in Figure 3.1 can be captured using predicates, as follows.

Definition 3.3.2. *Given the alphabet* $\Sigma = \{A, W, R, D\}$*, a sentence is valid if it satisfies the following properties.*

- *No R, W or D occurs before A. (Declaration)*
- *After an A, a W must occur before any R. (Initialization)*
- *After a D, an A must occur before any R, D, W. (Deletion)*
- *After an A, R, or W, a D must occur before an A. (Addition)*

For example, $AWWR$ is well-declared, whereas WWR is not. AWR is well-initialized, whereas ARW is not. $AWDAW$ is well-deleted, whereas $AWDDR$ is not. $AWDA$ is well-added whereas AWA is not. All the valid sentences are denoted by $\mathcal{L}(Valid)$.

Theorem 3.3.2. $\mathcal{L}(FSM_c) = \mathcal{L}(Valid)$.

Proof. It will be proven that given any sentence s over the alphabet Σ,

- If s can be accepted by FSM_c, then s is a valid sentence.
- If s is a valid sentence, then s can be accepted by FSM_c.

Table 3.1. Proof of "If $s.o$ can be accepted by FSM_c, then $s.o$ is a valid sentence". In the "s" part, each row shows an accepting state and its corresponding sentences. "State" means "accepting state". The "$s.o$" part shows the valid moves and their result sentences. Given a state and an input, if the combination is defined by Definition 3.3.1, then the resulting sentence is shown. An "x" indicates that a combination is not defined by Definition 3.3.1. For each non-x entry, it is verified against the four properties in Definition 3.3.2 on the assumption that s is a valid sentence.

s		$s.o$			
State	s	$o = A$	$o = W$	$o = R$	$o = D$
0	ϵ	$\epsilon.A$	x	x	x
	$\sim D$	$\sim D.A$	x	x	x
1	$\sim A$	x	$\sim A.W$	x	$\sim A.D$
2	$\sim W$	x	$\sim W.W$	$\sim W.R$	$\sim W.D$
	$\sim R$	x	$\sim R.W$	$\sim R.R$	$\sim R.D$

This done by using structure induction on s. **Base case.** By the definitions, an empty sentence ϵ can be accepted by FSM_c and is a valid sentence. **Inductive step.** It needs to be proven that given any $o \in \Sigma$,

1. If $s.o$ can be accepted by FSM_c, then $s.o$ is a valid sentence.
2. If $s.o$ is a valid sentence, then $s.o$ can be accepted by FSM_c.

on the inductive hypothesis (I.H.) that s can be accepted by FSM_c if and only if s is valid.

(1) Assume that $s.o$ can be accepted by FSM_c. The accepting state of s is any one of 0, 1 or 2. If the accepting state is 0, either s is an empty sentence ϵ or s ends with D. Note that $\sim D$ is used to indicate that s ends with D. Likewise, $\sim A$, $\sim W$, and $\sim R$ are used to indicate that s ends with A, W, and R, respectively.

- Case "s is ϵ". According to Definition 3.3.1, $s.o$ can be accepted only if o is A. In this case, if $s.o$ can be accepted, then $s.o$ must be A. According to Definition 3.3.2, A is a valid sentence.
- Case "s ends with D". According to Definition 3.3.1, $s.o$ can be accepted only if o is A. So if $s.o$ can be accepted, $s.o$ must be $\sim D.A$. According to I.H. that $\sim D$ is a valid sentence, appending A to $\sim D$ does not violate any of the four properties in Definition 3.3.2. So $\sim D.A$ is a valid sentence.

Likewise, the rest of the proof, in other cases where the accepting state is 1 or 2, can be obtained. The whole proof is summarized in Table 3.1.

(2)Assume that $s.o$ is a valid sentence. s can be accepted by FSM_c.

- If s is an empty sentence ϵ, only $s.A$ is valid by Definition 3.3.2. ($s.R$, $s.W$ and $s.D$ violate the "declaration" requirement.) According to Definition 3.3.1, $s.A$ can be accepted by FSM_c.

Table 3.2. Proof of "If $s.o$ is a valid sentence, then $s.o$ can be accepted by FSM_c". In the s part, each row shows a category of valid sentences which end with a common element. The accepting state for each category is also indicated. "State" means "accepting state". The $s.o$ part shows whether $s.o$ is valid, given a combination of s and o. If $s.o$ is valid, the correcting move is indicated. An "x" indicates an invalid combination of s and o. Invalid combinations of s and o can not be accepted by FSM_c.

s		$s.o$			
s	State	$o = A$	$o = W$	$o = R$	$o = D$
ϵ	0	$move(0, A)$	x	x	x
$\sim D$	0	$move(0, A)$	x	x	x
$\sim A$	1	x	$move(1, W)$	x	$move(1, D)$
$\sim W$	2	x	$move(2, W)$	$move(2, R)$	$move(2, D)$
$\sim R$	2	x	$move(2, W)$	$move(2, R)$	$move(2, D)$

- If s ends with D, only $s.A$ is valid, according to Definition 3.3.2. ($s.R$, $s.W$ and $s.D$ violate the "deletion" requirement.) Because of I.H. that s can be accepted by FSM_c, the accepting state of s is unique and must be 0, according to Lemma 3.3.2. So the sequence of moves of accepting s, with $move(0, A)$ appended at the end, can accept $s.A$.
- Likewise, the rest of the proof, in other cases where s ends with A, W, or R, can be obtained.

The whole proof is summarized in Table 3.2. $\square$

This theorem shows the equivalence of using the finite-state-machine approach and the predicate approach in specifying the validity constraint requirements. The finite-state-machine approach can be easily implemented in system design and implementation, while the predicate approach is largely adopted in system specification for correctness proof.

So far, validity requirements have been expressed on logs whose entries merely record operations on a single key of an data item. Next, the validity constraints will be extended to normal logs which record operations on the keys of all data items of a data space.

Property 3.3.1 (Declaration). *No read, write, or delete on a key is allowed before an add operation on the same key.*

Property 3.3.2 (Initialization). *After an add operation on a key, no read operation on the key is allowed before a write operation on the same key.*

Property 3.3.3 (Deletion). *After a delete operation on a key, no read, write or delete on the key is allowed before an add operation on the same key.*

Property 3.3.4 (Addition). *After an add, read, or write operation on a key, no add operation on the key is allowed before a delete operation on the same key.*

These properties can be formally expressed as predicates, as follows.

$$\begin{array}{|l}
\mathit{declared}\_ : \mathbb{P}(\mathrm{seq}\,\mathit{LogEntry}) \\
\hline
\forall\, l : \mathrm{seq}\,\mathit{LogEntry} \bullet \mathit{declared}(l) \Leftrightarrow \\
\quad \forall\, k \in \mathit{KEY};\ i : \mathrm{dom}\,\mathit{log} \bullet \mathit{key}(\mathit{log}(i)) = k \land (\mathit{op}(\mathit{log}(i)) = \mathit{read} \lor \\
\quad \mathit{op}(\mathit{log}(i)) = \mathit{write} \lor \mathit{op}(\mathit{log}(i)) = \mathit{delete}) \Rightarrow \\
\qquad \exists\, j : \mathrm{dom}\,\mathit{log} \bullet j < i \land \mathit{key}(\mathit{log}(j)) = k \land \\
\qquad\quad \mathit{op}(\mathit{log}(j)) = \mathit{add}
\end{array}$$

$$\begin{array}{|l}
\mathit{initialized}\_ : \mathbb{P}(\mathrm{seq}\,\mathit{LogEntry}) \\
\hline
\forall\, l : \mathrm{seq}\,\mathit{LogEntry} \bullet \mathit{initialized}(l) \Leftrightarrow \\
\quad \forall\, k \in \mathit{KEY};\ i : \mathrm{dom}\,\mathit{log} \bullet \mathit{key}(\mathit{log}(i)) = k \land \mathit{op}(\mathit{log}(i)) = \mathit{add} \Rightarrow \\
\quad (\forall\, j_1 : \mathrm{dom}\,\mathit{log} \bullet i < j_1 \land \mathit{key}(\mathit{log}(j_1)) = k \land \\
\qquad \mathit{op}(\mathit{log}(j_1)) = \mathit{read} \Rightarrow \\
\qquad\quad \exists\, j_2 : \mathrm{dom}\,\mathit{log} \bullet i < j_2 \land j_2 < j_1 \land \\
\qquad\qquad \mathit{key}(\mathit{log}(j_2)) = k \land \mathit{op}(\mathit{log}(j_2)) = \mathit{write})
\end{array}$$

$$\begin{array}{|l}
\mathit{deleted}\_ : \mathbb{P}(\mathrm{seq}\,\mathit{LogEntry}) \\
\hline
\forall\, l : \mathrm{seq}\,\mathit{LogEntry} \bullet \mathit{deleted}(l) \Leftrightarrow \\
\quad \forall\, k \in \mathit{KEY};\ i : \mathrm{dom}\,\mathit{log} \bullet \mathit{key}(\mathit{log}(i)) = k \land \mathit{op}(\mathit{log}(i)) = \mathit{delete} \Rightarrow \\
\quad (\forall\, j_1 : \mathrm{dom}\,\mathit{log} \bullet i < j_1 \land \mathit{key}(\mathit{log}(j_1)) = k \land \\
\qquad (\mathit{op}(\mathit{log}(j_1)) = \mathit{read} \lor \mathit{op}(\mathit{log}(j_1)) = \mathit{write} \lor \\
\qquad \mathit{op}(\mathit{log}(j_1)) = \mathit{delete}) \Rightarrow \\
\qquad\quad \exists\, j_2 : \mathrm{dom}\,\mathit{log} \bullet i < j_2 \land j_2 < j_1 \land \\
\qquad\qquad \mathit{key}(\mathit{log}(j_2)) = k \land \mathit{op}(\mathit{log}(j_2)) = \mathit{add})
\end{array}$$

$$\begin{array}{|l}
\mathit{added}\_ : \mathbb{P}(\mathrm{seq}\,\mathit{LogEntry}) \\
\hline
\forall\, l : \mathrm{seq}\,\mathit{LogEntry} \bullet \mathit{added}(l) \Leftrightarrow \\
\quad \forall\, k \in \mathit{KEY};\ i : \mathrm{dom}\,\mathit{log} \bullet \mathit{key}(\mathit{log}(i)) = k \land \\
\quad (\mathit{op}(\mathit{log}(i)) = \mathit{add} \lor \mathit{op}(\mathit{log}(i)) = \mathit{read} \lor \mathit{op}(\mathit{log}(i)) = \mathit{write}) \Rightarrow \\
\quad (\forall\, j_1 : \mathrm{dom}\,\mathit{log} \bullet i < j_1 \land \mathit{key}(\mathit{log}(j_1)) = k \land \\
\qquad \mathit{op}(\mathit{log}(j_1)) = \mathit{add} \Rightarrow \\
\qquad\quad \exists\, j_2 : \mathrm{dom}\,\mathit{log} \bullet i < j_2 \land j_2 < j_1 \land \\
\qquad\qquad \mathit{key}(\mathit{log}(j_2)) = k \land \mathit{op}(\mathit{log}(j_2)) = \mathit{delete})
\end{array}$$

So the validity of a log can be defined by using these required properties combined.

$$\begin{array}{|l}
\mathit{isValidLog}\_ : \mathbb{P}(\mathrm{seq}\,\mathit{LogEntry}) \\
\hline
\forall\, l : \mathrm{seq}\,\mathit{LogEntry} \bullet \mathit{isValidLog}(l) \Leftrightarrow \\
\quad \mathit{declared}(l) \land \mathit{initialized}(l) \land \mathit{deleted}(l) \land \mathit{added}(l)
\end{array}$$

The full specification of valid logs is given by combining the property-based specifications developed above.

$$\mathit{ValidLog} == \{l : \mathit{Log} \mid \mathit{isValidLog}(l)\}$$

3.3.2 Logs of data spaces

In a DU system, logs are used to record data accesses on data spaces. In order to give a formal specification of a DU system using schema calculus, it is necessary to provide schematic specification of logs of data spaces.

The log of a data space is modelled by the following schema.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{DSLog}} \\
\hline
\mathit{log} : \mathit{Log} \\
\hline
\end{array}$$

In this schema, the above-mentioned data type *Log* is used as the data structure for capturing data accessed. Initially, a log has no log entries. The *DSLogInit* schema models a log in its initial state.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{DSLogInit}} \\
\hline
\mathit{log} \\
\hline
\mathit{log} = \langle\rangle \\
\hline
\end{array}$$

The schemas *ΔDSLog* and *ΞDSLog* will be used when operations on logs are specified. If an operation changes the state of a log, *ΔDSLog* is included in the specification of the operation; otherwise, *ΞDSLog* is used.

$$\Delta \mathit{DSLog} \mathrel{\widehat=} [\mathit{DSLog};\ \mathit{DSLog}']$$
$$\Xi \mathit{DSLog} \mathrel{\widehat=} [\Delta \mathit{DSLog} \mid \theta \mathit{DSLog} = \theta \mathit{DSLog}']$$

Since logs are used for recording accesses on data spaces, logs are changed per access. Such changes can be explicitly modelled. With respect to the four operations on data spaces, four operations on logs are defined.

Adding a data item to a data space results in a new log entry appended to the log of the give data space. The *append* function defined above is used here to update the state of the variable *log*. Before an add operation, the specified data item must not exist in the given data space. Furthermore, the value of the specified data item before the add operation can be any value.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{AppendAdd}} \\
\hline
\Delta \mathit{DSLog} \\
\Xi \mathit{DataSpace} \\
t? : \mathit{TIME} \\
k? : \mathit{KEY} \\
v? : \mathit{VAL} \\
\hline
k? \notin \operatorname{dom} \mathit{content} \\
\exists\, w : \mathit{VAL} \bullet \mathit{log}' = \mathit{append}(\mathit{log}, ((((t?, \mathit{add}), k?), w), v?)) \\
\hline
\end{array}$$

Reading the value of a data item from a data space introduces a log appending, as follows.

$$
\begin{array}{|l}
\textit{AppendRead} \\
\Delta \textit{DSLog} \\
\Xi \textit{DataSpace} \\
t? : \textit{TIME} \\
k? : \textit{KEY} \\
\hline
k? \in \operatorname{dom} \textit{content} \\
\textit{log}' = \textit{append}(\textit{log}, ((((t?, \textit{read}), k?), \textit{content}(k?)), \textit{content}(k?)))
\end{array}
$$

Writing a new value to a data item introduces the following state modification to *log*.

$$
\begin{array}{|l}
\textit{AppendWrite} \\
\Delta \textit{DSLog} \\
\Xi \textit{DataSpace} \\
t? : \textit{TIME} \\
k? : \textit{KEY} \\
v? : \textit{VAL} \\
\hline
k? \in \operatorname{dom} \textit{content} \\
\textit{log}' = \textit{append}(\textit{log}, ((((t?, \textit{write}), k?), \textit{content}(k?)), v?))
\end{array}
$$

Deleting a data item from a data space changes the state of *log*, as specified by the *AppendDelete* schema. After a delete operation, the specified data item will no longer exist in the given data space. So the value of the specified data item after the delete operation can be any value.

$$
\begin{array}{|l}
\textit{AppendDelete} \\
\Delta \textit{DSLog} \\
\Xi \textit{DataSpace} \\
t? : \textit{TIME} \\
k? : \textit{KEY} \\
\hline
k? \in \operatorname{dom} \textit{content} \\
\exists\, w : \textit{VAL} \bullet \textit{log}' = \textit{append}(\textit{log}, ((((t?, \textit{delete}), k?), \textit{content}(k?)), w))
\end{array}
$$

3.4 System specification

A DU system consists of a number of data spaces. Data spaces contain data items and their copies. Data items can be added to or removed from data spaces. Data items stored in data spaces can be read or modified. Furthermore, data spaces can be synchronized to ensure that a given data item has the same value in the data spaces.

To specify a DU system, one additional basic type is introduced to identify data spaces. LOC is a set of identifiers that denote data spaces.

$$[LOC]$$

A DU system is modelled as a pair of partial functions. *spaces* models the states of data spaces and *logs* models the logs of data spaces. It is assumed that each data space has an associated log. Assume that each data space is uniquely identified in the system. The state and log of a data space can be retrieved by the identifier of the data space.

$$\begin{array}{|l}
\textit{System} \\
\hline
\textit{spaces} : LOC \text{ ⇸ } \textit{DataSpace} \\
\textit{logs} : LOC \text{ ⇸ } \textit{DSLog} \\
\hline
\operatorname{dom} \textit{spaces} = \operatorname{dom} \textit{logs} \\
\hline
\end{array}$$

It is assumed that a DU system is static in its composition, although it is possible to model dynamics of a DU system, such as adding or removing data spaces. The component data spaces of a DU system can be obtained by applying the domain function "dom" to *spaces*. According to this assumption, the component data spaces of a DU system do not change over time, whereas the states of the component data spaces may be changed by operations.

The following pair of schemas will be used when describing system operations. If an operation changes the state of the system, $\Delta \textit{System}$ will be included in the specification of the operation; otherwise, $\Xi \textit{System}$ will be included.

$$\Delta \textit{System} \mathrel{\widehat{=}} [\textit{System};\ \textit{System}']$$
$$\Xi \textit{System} \mathrel{\widehat{=}} [\Delta \textit{System} \mid \theta \textit{System} = \theta \textit{System}']$$

To define the system operations on data spaces, it is necessary to factor a system operation into operations on some data spaces by expressing the relationship between the state of a system and that of the system component. The technique of *promotion* [174] is used here.

The promotion from operations on data spaces to those on the system is characterized by the *Promote* schema.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_\,\mathit{Promote}\,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta \mathit{System} \\
\Delta \mathit{DataSpace} \\
l? : \mathit{LOC} \\
\hline
l? \in \operatorname{dom} \mathit{spaces} \\
\theta \mathit{DataSpace} = \mathit{spaces}(l?) \\
\theta \mathit{DataSpace}' = \mathit{spaces}'(l?) \\
\{l?\} \ntriangleleft \mathit{spaces} = \{l?\} \ntriangleleft \mathit{spaces}' \\
\{l?\} \ntriangleleft \mathit{logs} = \{l?\} \ntriangleleft \mathit{logs}' \\
\hline
\end{array}
$$

To complete the specification, the *REPORT* type is extended to include another value *not_in_system*.

$$\mathit{REPORT} ::= \mathit{key\_in\_use} \mid \mathit{key\_not\_in\_use} \mid \mathit{success} \mid \mathit{not\_in\_system}$$

The value *not_in_system* indicates an error in an operation, saying that the operated data space is not within the system. The *NotInSystem* schema is used to specify such an unsuccessful operation.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_\,\mathit{NotInSystem}\,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Xi \mathit{DataSpace} \\
l? : \mathit{LOC} \\
\mathit{result}! : \mathit{REPORT} \\
\hline
l? \notin \operatorname{dom} \mathit{spaces} \\
\mathit{result}! = \mathit{not\_in\_system} \\
\hline
\end{array}
$$

The system operations are defined as follows, using the *Promote* schema. Note that only successful accesses are logged.

$$
\begin{array}{ll}
\mathit{SAdd} & \widehat{=}\ (\exists\, \Delta \mathit{DataSpace};\ \Delta \mathit{DSLog} \bullet (\mathit{DAdd}_0 \land \mathit{AppendAdd} \land \mathit{Success} \lor \\
& \quad \mathit{KeyInUse}) \land \mathit{Promote}) \lor \mathit{NotInSystem} \\
\mathit{SRead} & \widehat{=}\ (\exists\, \Delta \mathit{DataSpace};\ \Delta \mathit{DSLog} \bullet (\mathit{DRead}_0 \land \mathit{AppendRead} \land \\
& \quad \mathit{Success} \lor \mathit{KeyNotInUse}) \land \mathit{Promote}) \lor \mathit{NotInSystem} \\
\mathit{SWrite} & \widehat{=}\ (\exists\, \Delta \mathit{DataSpace};\ \Delta \mathit{DSLog} \bullet (\mathit{DWrite}_0 \land \mathit{AppendWrite} \land \\
& \quad \mathit{Success} \lor \mathit{KeyNotInUse}) \land \mathit{Promote}) \lor \mathit{NotInSystem} \\
\mathit{SDelete} & \widehat{=}\ (\exists\, \Delta \mathit{DataSpace};\ \Delta \mathit{DSLog} \bullet (\mathit{DDelete}_0 \land \mathit{AppendDelete} \land \\
& \quad \mathit{Success} \lor \mathit{KeyNotInUse}) \land \mathit{Promote}) \lor \mathit{NotInSystem}
\end{array}
$$

To manifest the correctness of this promotion, *SWrite* can for example be unfolded to the *AnotherSWrite* schema without promotion.

```
— AnotherSWrite ————————————————————————
| $\Delta System$
| $l? : LOC$
| $k? : KEY$
| $v? : VAL$
| $t? : TIME$
| $result! : REPORT$
|————————
| $(l? \in \operatorname{dom} spaces \Rightarrow$
|       $\textbf{if } k? \in \operatorname{dom} spaces(l?).content$
|       $\textbf{then } \{l?\} \lhd spaces' = \{l?\} \lhd spaces \land$
|             $\{l?\} \lhd logs' = \{l?\} \lhd logs \land$
|             $spaces'(l?).content = spaces(l?).content \oplus \{k? \mapsto v?\} \land$
|             $logs'(l?) = append(logs(l?),$
|                   $((((t?, write), k?), spaces(l?).content(k?)), v?)) \land$
|             $result! = success$
|       $\textbf{else } spaces' = space \land logs' = logs \land$
|             $result! = key\_not\_in\_use)$
| $(l? \notin \operatorname{dom} spaces \Rightarrow spaces' = space \land logs' = logs \land$
|       $result! = not\_in\_system)$
————————————————————————————————————————
```

3.4.1 Consistency

Of the above-mentioned operations, the *SAdd* operation can be used to replicate or copy a data item from one data space to another. The values of different copies of the data item can subsequently be updated by using the *SWrite* operation in different data spaces. Such an update can be performed regardless of its copies in other data spaces. Updating data items in a disconnected manner may introduce inconsistencies.

With respect to data consistency in Definition 3.1.4, *DConsistent* is defined to check whether a given data item has the same value in two given data spaces.

```
| $DConsistent : \mathbb{P}(DataSpace \times DataSpace \times KEY)$
|————————
| $\forall S_1, S_2 : DataSpace;\ k : KEY \bullet DConsistent(S_1, S_2, k) \Leftrightarrow$
|       $(k \in \operatorname{dom}(S_1.content) \land k \in \operatorname{dom}(S_2.content) \Rightarrow$
|             $S_1.content(k) = S_2.content(k))$
```

With respect to system consistency in Definition 3.1.5, *SConsistent* is defined to determine whether a DU system is consistent or not.

```
| $SConsistent : \mathbb{P}(System)$
|————————
| $\forall s : System \bullet SCconsistent(s) \Leftrightarrow (\forall k : KEY;\ l_1, l_1 : \operatorname{dom}(s.spaces) \bullet$
|       $DConsistent(s.spaces(l_1), s.spaces(l_2), k))$
```

Note that consistency depends merely on the current values of data items in the system. It does not take into account logs. Logs are only used in data synchronization.

3.4.2 Data synchronization

When a DU system becomes inconsistent, data synchronization can be performed on request. *SSync* provides a formal specification of the synchronization operation.

$$
\begin{array}{|l}
\hline
\textit{SSync} \\
\Delta \textit{System} \\
l_1?, l_2? : \textit{LOC} \\
k? : \textit{KEY} \\
\textit{result}! : \textit{REPORT} \\
\hline
l_1? \in \operatorname{dom} \textit{spaces} \\
l_2? \in \operatorname{dom} \textit{spaces} \\
l_1? \neq l_2? \\
((k? \in \operatorname{dom} \textit{spaces}(l_1?).\textit{content} \land k? \notin \operatorname{dom} \textit{spaces}(l_2?).\textit{content}) \lor \\
\quad (k? \notin \operatorname{dom} \textit{spaces}(l_1?).\textit{content} \land k? \in \operatorname{dom} \textit{spaces}(l_2?).\textit{content}) \lor \\
\quad (k? \in \operatorname{dom} \textit{spaces}(l_1?).\textit{content} \land k? \in \operatorname{dom} \textit{spaces}(l_2?).\textit{content} \land \\
\quad \textit{spaces}(l_1?).\textit{content}(k?) \neq \textit{spaces}(l_2?).\textit{content}(k?))) \\
\{l_1?, l_2?\} \lhd \textit{spaces}' = \{l_1?, l_2?\} \lhd \textit{spaces} \\
\{k?\} \lhd \textit{spaces}'(l_1?).\textit{content} = \{k?\} \lhd \textit{spaces}(l_1?).\textit{content} \\
\{k?\} \lhd \textit{spaces}'(l_2?).\textit{content} = \{k?\} \lhd \textit{spaces}(l_2?).\textit{content} \\
\textit{spaces}'(l_1?).\textit{content}(k?) = \textit{spaces}'(l_2?).\textit{content}(k?) \\
\textit{result}! = \textit{success} \\
\hline
\end{array}
$$

The precondition of the operation is the conjunction of the following.

- $l_1?$ is in the system.
- $l_2?$ is in the system.
- $l_1?$ is not the same as $l_2?$.
- $k?$ is defined in either of the data spaces. If $k?$ is defined in both data spaces, the data item has different values in the data spaces.

The postcondition of the operation is the conjunction of the following.

- The rest of the system, i.e. data spaces other than the two denoted by $l_1?$ and $l_2?$, remains unchanged.
- In the data spaces denoted by $l_1?$ and $l_2?$, the data item denoted by $k?$ has the same value while the others remain unchanged.

3.5 Concluding remarks

In this chapter a model of DU systems has been formalized. In this model, storage devices are modelled as data spaces, data are modelled as data items, and data replication is also modelled. Disconnected updates are modelled as operations that modify data items independent of their copies. Logs are modelled as access histories of data spaces. Consistency criteria have been established. A formal specification of data synchronization has also been presented.

In models of distributed databases systems [23], a distinction is made between logical data items and physical data items: logical data items are observed by the users; logical data items are implemented by a set of physical data items distributed over storage devices; and physical data items are not visible to the users. In comparison with those models, the concept of data items in the model of DU systems is closer to that of physical data items in those models. The notion of logical data items is discarded for the following reasons. In the first place, the property of location-transparency is applicable in traditional distributed systems with reliable network connections, but not in living environments with dynamic network connections.[8] Secondly, users conceptually associate data with locations (devices) and tend to have data copies stored in different locations (devices). Thirdly, this treatment allows to focus mainly on resolving inconsistencies of data copies, instead of on the mapping between logical data and physical data.

[8] Issues relating to locations will be discussed in Section 6.1.

4. Characteristic-entry logs

Characteristic-entry logs record only the most recent access of each operation type. In this chapter, an informal description of characteristic-entry logs will first be given described. Next, characteristic-entry logs will be formally modelled. After that, the relation between normal logs and characteristic-entry logs will be investigated. Finally, a formal characterization will be defined, which converts a normal log into a characteristic-entry log.

4.1 Motivation

In most logging mechanisms, log size is restricted due to storage constraints. Sometimes log size may also be deliberately limited by system administrators. A log file grows linearly with the number of data accesses performed. When the size of a log file reaches its limit, any new data access will cause log overflow. This occurs especially in portable computing devices with memory and storage constraints.

To solve the problem of log overflow in practice system administrators will be notified of such risks and will take measures to prevent them, such as manually compressing logs, enlarging the limit of log size or deleting log entries. Otherwise, systems won't be able to track further accesses. In professional environments, such administrative tasks are executed by dedicated skilled persons with professional knowledge. It is not reasonable to assume that persons capable of and responsible for those tasks will be available in home environments. Logging systems requiring minimum administrative efforts are desired for distributed systems in home environments.

Information loss in log truncation. An alternative solution to log overflow is to discard new entries when log files reach their size limits. When this approach is used, logs will contain only the oldest accesses. Another solution to log overflow is to discard the oldest entries in logs when logs reach their limitation. In this approach, logs will contain only the most recent accesses. In both cases, useful information may be lost when log entries are discarded.

Consider for example a data space S and a data item x. S can be a personal digital assistant and x can be a file. Let a log file l record all data accesses on the data space S. Assume that the size of l will never exceed 4, meaning that at most four accesses can be recorded. Suppose the following sequence of accesses will be consecutively performed on S, from left to right.

$$A(x),\, W(x)1, R(x)1,\, W(x)2, R(x)2, R(x)2, R(x)2, R(x)2.$$

For simplicity, timestamps have been omitted from the log entries.

If new entries are discarded in the case of log overflow, log l will look as follows after completion of the last access of the access sequence.

$$l = \langle A(x),\, W(x)1, R(x)1,\, W(x)2\rangle.$$

This log does not show the last time x was read. Read accesses can be of relevance in synchronizing data copies. For instance, suppose that another copy of x was deleted from the device T right after the $W(x)2$ access and before the first $R(x)2$ access on the device S. If the user expects that the most recent operation will be applied to all copies of x in synchronization, the deletion should be ignored. Because l does not reveal the four read accesses that occurred after the deletion, the deletion will in this case be applied to S, which will result in data loss. Therefore, a read access on a data item cannot be simply replaced by the most recent write access on the same data item.

If the oldest entries are discarded in the case of log overflow, log l will look as follows after completion of the last access of the the access sequence.

$$l = \langle R(x)2, R(x)2, R(x)2, R(x)2\rangle.$$

This log does not tell when x was added and when it was modified. In both cases, the discarded log entries contain information that may be of use in data synchronization.

Uneven distribution of log spaces. Logs with size limits may contain only accesses of the most-frequently-visited data items while less-frequently-used data become "historyless". In other words, log spaces are not evenly distributed according to data items. Suppose that the following sequence of accesses will be applied to S. Again, assume the size limit of l is 4.

$$A(x),\, W(x)1, R(x)1, A(y),\, W(y)2, R(y)2, D(y).$$

So after completion of the last access, log l will look as follows according to the method of discarding the oldest entries in the case of log overflow.

$$l = \langle A(y),\, W(y)2, R(y)2, D(y)\rangle.$$

In this case, l contains only access information on y. It contains no information on x whatsoever, which makes it difficult to synchronize x with its copies.

Resource-constrained log exchange. Besides log overflow, actual log size may affect system performance during data synchronization. Exchanging lengthy log files between portable devices or between portable devices and stationary desktop computers is constrained by the bandwidth of network connections and the computing power of portable devices. Data synchronization can become a rather slow process.

End-users always welcome synchronization that can provide instant results. More efficient ways of exchanging log files are needed. In practice, log files are often truncated to minimize the amount of data to be transmitted over networks. Truncating logs without precaution might introduce the information-loss and the uneven-distribution problems described above.

Characteristic-entry logs. The above examples make it clear that size limits of log files are of great importance in recording access information. Truncating logs may cause trouble in data synchronization. It is also shown that log spaces are not evenly distributed, which may also introduce difficulties in data synchronization. In practice, size limits are often set as large as possible and are usually only constrained by storage capacities of devices. This approach can however not be followed in the case of mobile computing systems due to storage capacity limits of mobile hand-held devices and limited bandwidth of network connections. Setting realistic log limits is an empirical task. Any realistic estimate of log size has to be derived from empirical data. With wrong estimations, information loss in logs is inevitable, making data synchronization vulnerable.

Characteristic-entry logs record only the most recent add, read, write and delete accesses for each data item [122]. Repetitive accesses are not recorded in logs. In this way, log size is no longer determined by the number of accesses, but by the number of data items and the number of access types. Therefore, log sizes become predicable and manageable. Moreover, log space is evenly distributed over all data items. Figure. 4.1 illustrates an example. In the figure, a sequence of accesses on the data space S is shown horizontally, with a time line running from left to right. Each access is specified by its operation type and the value of the accessed data item. The normal log and the characteristic-entry logs of S are illustrated. For simplicity, timestamps have been omitted from the log entries.

Given a data space, the size of a characteristic-entry log is bounded by the product of the number of data items and the number of operation types. Meanwhile, characteristic-entry logs can still be used for data synchronization in the same way as normal logs, as will be shown in Chapter 5. In the subsequent sections of this chapter characteristic-entry logs will be formalized. It will be shown that characteristic-entry logs can be converted from normal logs and that characteristic-entry logs can be constructed at run time.

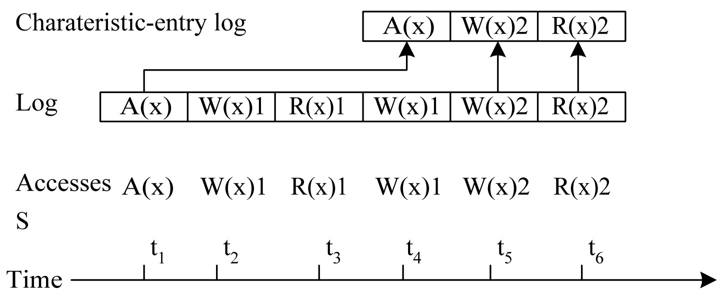


Figure 4.1. A normal log and its corresponding characteristic-entry log. The characteristic-entry log records only the most recent accesses of each operation type.

4.2 Formalizing characteristic-entry logs

A characteristic-entry log records merely the last operation of each operation type on a key. In other words, given a key and an operation type, there exists at most one log entry with the key and the operation type in a characteristic-entry log. This property is specified by the predicate *isCEL*.

$$
\begin{array}{|l}
\mathit{isCEL}\_ : \mathbb{P}(\mathrm{seq}\,\mathit{LogEntry}) \\
\hline
\forall\, l : \mathrm{seq}\,\mathit{LogEntry} \bullet \mathit{isCEL}(l) \Leftrightarrow \\
\quad \forall\, i, j : \mathrm{dom}\, l \bullet (i \neq j \land \mathit{key}(l(i)) = \mathit{key}(l(j))) \Rightarrow \\
\qquad \mathit{op}(l(i)) \neq \mathit{op}(l(j))
\end{array}
$$

Thus, characteristic-entry logs can be formally specified as follows.

$$\mathit{CELog} == \{l : \mathit{Log} \mid \mathit{isCEL}(l)\}.$$

This definition indicates only that there is at most one entry for an operation of a type on a key. It does not say that a recorded access is the last one of this sort. This "most recentness" property is ensured by the appending function of characteristic-entry logs.

Before a log entry is appended to a characteristic-entry log, the log is checked to see whether it already contains an element that has the same key and operation type as that of the log entry to be appended. If so, the element must be removed from the log. This can be done with the *delete* function.

$$
\begin{array}{|l}
\mathit{delete} : \mathrm{seq}\,\mathit{LogEntry} \times (\mathit{OP} \times \mathit{KEY}) \rightarrow \mathrm{seq}\,\mathit{LogEntry} \\
\hline
\mathit{delete} = \lambda\, l : \mathrm{seq}\,\mathit{LogEntry};\ \mathit{ok} : (\mathit{OP} \times \mathit{KEY}) \bullet \\
\quad \textbf{if } l = \langle\rangle \textbf{ then } \langle\rangle \\
\quad \textbf{else if } \mathit{op}(\mathit{last}(l)) = \mathit{first}(\mathit{ok}) \land \mathit{key}(\mathit{last}(l)) = \mathit{second}(\mathit{ok}) \\
\qquad \textbf{then } \mathit{delete}(\mathit{front}(l), \mathit{ok}) \\
\qquad \textbf{else } \mathit{delete}(\mathit{front}(l), \mathit{ok}) \frown \langle \mathit{last}(l) \rangle
\end{array}
$$

The *delete* function works backward on a log.[1] This is because, in practice, successive operations tend to access the same data item. So it is more efficient to locate a matching log entry, with the specified key and operation, from the tail of the log to its head.

Using *delete*, the *concat* function is defined for appending a log entry to a characteristic-entry log. This function ensures that a characteristic-entry log records only the last operation of each type on a key.

```
$concat : \text{seq}\, LogEntry \times LogEntry \rightarrow \text{seq}\, LogEntry$
----
$concat = \lambda\, l : \text{seq}\, LogEntry;\; e : LogEntry \bullet$
    $\textbf{if}\ isAfterLast(l, time(e))\ \textbf{then}\ delete(l, (op(e), key(e))) \frown \langle e \rangle$
    $\textbf{else}\ l$
```

For example, assume a normal log l which records a sequence of accesses on a data item, where $l = \langle A;\; W_1;\; R \rangle$. Note that in the notation it is assumed that all the operations are related to a common key. The key, execution time, pre-value of the key and post-value of the key are all omitted for simplicity. Only the operations of the entries of the log are shown. The operations are ordered by time. The right-most operation is assumed to be the last performed. Now, consider a log entry W_2 to be appended to l, assuming that $isAfterLast(l, time(W_2))$ holds. Appending the log entry to the log is illustrated as follows.

$$
\begin{aligned}
& concat(l, W_2) \\
= {} & delete(l, (op(W_2), key(W_2))) \frown \langle W_2 \rangle \\
= {} & delete(front(l), (op(W_2), key(W_2))) \frown \langle last(l) \rangle \frown \langle W_2 \rangle \\
= {} & delete(\langle A;\; W_1 \rangle, (op(W_2), key(W_2))) \frown \langle R \rangle \frown \langle W_2 \rangle \\
= {} & delete(front(\langle A;\; W_1 \rangle), (op(W_2), key(W_2))) \frown \langle R \rangle \frown \langle W_2 \rangle \\
= {} & delete(\langle A \rangle, (op(W_2), key(W_2))) \frown \langle R \rangle \frown \langle W_2 \rangle
\end{aligned}
$$

[1] According to the definition, the *delete* function removes all the elements with the specified key and operation type from a log entry sequence by scanning the entire sequence from its end to its beginning. This *delete* function works for any log entry sequence even if the given sequence does not preserve the *isCEL* property. When the given log entry sequence preserves the *isCEL* property, the *delete* function can be further simplified to the following form.

```
$delete : \text{seq}\, LogEntry \times (OP \times KEY) \rightarrow \text{seq}\, LogEntry$
----
$delete = \lambda\, l : \text{seq}\, LogEntry;\; ok : (OP \times KEY) \bullet$
    $\textbf{if}\ l = \langle\rangle\ \textbf{then}\ \langle\rangle$
    $\textbf{else if}\ op(last(l)) = first(ok) \land key(last(l)) = second(ok)$
        $\textbf{then}\ front(l)$
        $\textbf{else}\ delete(front(l), ok) \frown \langle last(l) \rangle$
```

$$
\begin{aligned}
&= \mathit{delete}(\mathit{front}(\langle A\rangle), (\mathit{op}(W_2), \mathit{key}(W_2))) \frown \mathit{last}(\langle A\rangle) \frown \langle R\rangle \frown \langle W_2\rangle \\
&= \mathit{delete}(\langle\rangle, (\mathit{op}(W_2), \mathit{key}(W_2))) \frown \langle A\rangle \frown \langle R\rangle \frown \langle W_2\rangle \\
&= \langle\rangle \frown \langle A\rangle \frown \langle R\rangle \frown \langle W_2\rangle \\
&= \langle A\rangle \frown \langle R\rangle \frown \langle W_2\rangle \\
&= \langle A;\ R\rangle \frown \langle W_2\rangle \\
&= \langle A;\ R;\ W_2\rangle
\end{aligned}
$$

4.2.1 Properties of log concatenation

When *concat* is used to concatenate a characteristic-entry log and a log entry, the result will still be a characteristic-entry log. Before this property is proven, the following properties of the *delete* operation need to be examined.

- Non-creation. The result of the *delete* operation contains no entries that were not present in the original log.
- Effectiveness. The result of the *delete* operation contains no entries matching the specified operation and key.
- No-time-shift. The result of the *delete* operation does not violate the timing of the original log.
- Order-preservation. The result of the *delete* operation preserves the order of the original log.

Lemma 4.2.1 (Non-creation of *delete*). *Given $l : \operatorname{seq} \mathit{LogEntry}$, $k : \mathit{KEY}$, and $o : \mathit{OP}$, $\forall i : \operatorname{dom} \mathit{delete}(l, (o, k)) \bullet \exists j : \operatorname{dom} l \bullet \mathit{delete}(l, (o, k))(i) = l(j)$.*

Proof. The proof is obtained using structure induction on l.[2] **Base case.** It needs to be proven that

$$\forall i : \operatorname{dom} \mathit{delete}(\langle\rangle, (o, k)) \bullet \exists j : \operatorname{dom}\langle\rangle \bullet \mathit{delete}(\langle\rangle, (o, k))(i) = \langle\rangle(j) \quad (4.1)$$

[2] There are several versions of structural induction on sequences, as described in [36]. One version says that in order to show that some property $P(l)$ holds for all sequences l, the following should be proven. **1.** $P(\langle\rangle)$ holds. **2.** If $P(l)$ holds for any sequence l, then so does $P(\langle x\rangle \frown l)$. Formally:

$$\forall x : X;\ l : \operatorname{seq} X \bullet P(l) \Rightarrow P(\langle x\rangle \frown l).$$

Another version says that in order to show that some property $P(l)$ holds for all sequences $l : \operatorname{seq} X$, the following should be proven. **1.** $P(\langle\rangle)$ holds. **2.** If $P(l)$ holds for any sequence l, then so does $P(l \frown \langle x\rangle)$. Formally:

$$\forall x : X;\ l : \operatorname{seq} X \bullet P(l) \Rightarrow P(l \frown \langle x\rangle).$$

Because the *delete* function is recursively defined by checking the last element of a sequence, the second version of structural induction on sequences have been chosen to prove properties of *delete* in this chapter.

$$
\begin{aligned}
&(4.1)\\
\equiv\ &\{\text{Definition of } \mathit{delete}\}\\
&\forall\, i : \mathrm{dom}\langle\rangle \bullet \exists\, j : \mathrm{dom}\langle\rangle \bullet \langle\rangle(i) = \langle\rangle(j)\\
\equiv\ &\mathit{true}
\end{aligned}
$$

This establishes the base case. **Inductive step.** It needs to be proven that

$$
\begin{aligned}
&\forall\, i :\ \mathrm{dom}\, \mathit{delete}(l \frown \langle e\rangle, (o,k)) \bullet \exists\, j : \mathrm{dom}\, l \frown \langle e\rangle \bullet\\
&\qquad \mathit{delete}(l \frown \langle e\rangle, (o,k))(i) = l \frown \langle e\rangle(j)
\end{aligned}
\tag{4.2}
$$

on the assumption that

$$
\forall\, i : \mathrm{dom}\, \mathit{delete}(l,(o,k)) \bullet \exists\, j : \mathrm{dom}\, l \bullet \mathit{delete}(l,(o,k))(i) = l(j) \qquad \text{(I.H.)}
$$

(I.H. stands for induction hypothesis.) If $op(e) = o \wedge key(e) = k$ holds,

$$
\begin{aligned}
&(4.2)\\
\equiv\ &\{\text{Definition of } \mathit{delete}\}\\
&\forall\, i : \mathrm{dom}\, \mathit{delete}(l,(o,k)) \bullet \exists\, j : \mathrm{dom}\, l \frown \langle e\rangle \bullet\\
&\qquad \mathit{delete}(l,(o,k))(i) = l \frown \langle e\rangle(j)\\
\Leftarrow\ &\{\text{let } j \text{ over } 1..\,\mathrm{dom}\, l \frown \langle e\rangle\}\\
&\text{I.H.}
\end{aligned}
$$

If $op(e) = o \wedge key(e) = k$ does NOT hold,

$$
\begin{aligned}
&(4.2)\\
\equiv\ &\{\text{Definition of } \mathit{delete}\}\\
&\forall\, i : \mathrm{dom}\, \mathit{delete}(l,(o,k)) \frown \langle e\rangle \bullet \exists\, j : \mathrm{dom}\, l \frown \langle e\rangle \bullet\\
&\qquad \mathit{delete}(l,(o,k)) \frown \langle e\rangle(i) = l \frown \langle e\rangle(j)\\
\equiv\ &\{\text{divide into two cases}\}\\
&(\forall\, i : \mathrm{dom}\, \mathit{delete}(l,(o,k)) \frown \langle e\rangle \bullet\\
&\qquad (i < \#(\mathit{delete}(l,(o,k)) \frown \langle e\rangle) \wedge\\
&\qquad\qquad \exists\, j : \mathrm{dom}\, l \frown \langle e\rangle \bullet \mathit{delete}(l,(o,k)) \frown \langle e\rangle(i) = l \frown \langle e\rangle(j)) \vee\\
&\qquad (i = \#(\mathit{delete}(l,(o,k)) \frown \langle e\rangle) \wedge\\
&\qquad\qquad \exists\, j : \mathrm{dom}\, l \frown \langle e\rangle \bullet \mathit{delete}(l,(o,k)) \frown \langle e\rangle(i) = l \frown \langle e\rangle(j))\\
\Leftarrow\ &\{\text{If } i < \#(\mathrm{dom}\, \mathit{delete}(l,(o,k)) \frown \langle e\rangle)\text{, there exists a } k \text{ such that}\\
&\qquad \mathit{delete}(l,(o,k))(i) = l(k) \text{ by I.H. Let } j = k.\\
&\qquad \text{If } i = \#(\mathrm{dom}\, \mathit{delete}(l,(o,k)) \frown \langle e\rangle)\text{, Let } j = \#(l \frown \langle e\rangle).\}\\
&\text{I.H.}
\end{aligned}
$$

This establishes the inductive step. Since both the base case and the inductive step have been shown to be true, it follows by the principle of structure induction on sequences that the lemma holds. □

Lemma 4.2.2 (Effectiveness of *delete*). *Given* $l : \mathrm{seq}\, LogEntry$*,* $k : KEY$*, and* $o : OP$*,* $\neg \exists\, i : \mathrm{dom}\, delete(l, (o, k)) \bullet op(delete(l, (o, k))(i)) = o \wedge key(delete(l, (o, k))(i)) = k$*.*

Proof. Structure induction on l. **Base case.** $l = \langle\rangle$: Trivial. **Inductive step.** It needs to be proven that

$$\begin{aligned}\neg \exists\, i :\ & \mathrm{dom}\, delete(l \frown \langle e\rangle, (o, k)) \bullet op(delete(l \frown \langle e\rangle, (o, k))(i)) = o \wedge \\ & key(delete(l \frown \langle e\rangle, (o, k))(i)) = k \end{aligned} \tag{4.3}$$

on the assumption that

$$\begin{aligned}\neg \exists\, i :\ & \mathrm{dom}\, delete(l, (o, k)) \bullet op(delete(l, (o, k))(i)) = o \wedge \\ & key(delete(l, (o, k))(i)) = k. \end{aligned} \tag{I.H.}$$

If $op(e) = o \wedge key(e) = k$ holds,

$$\begin{aligned}
& (4.3) \\
\equiv\ & \{\text{Definition of } delete\} \\
& \neg \exists\, i : \mathrm{dom}\, delete(l, (o, k)) \bullet op(delete(l, (o, k))(i)) = o \wedge \\
& \qquad key(delete(l, (o, k))(i)) = k \\
\equiv\ & \text{I.H.}
\end{aligned}$$

If $op(e) = o \wedge key(e) = k$ does NOT hold,

$$\begin{aligned}
& (4.3) \\
\equiv\ & \{\text{Definition of } delete\} \\
& \neg \exists\, i : \mathrm{dom}\, delete(l, (o, k)) \frown \langle e\rangle \bullet op(delete(l, (o, k)) \frown \langle e\rangle(i)) = o \wedge \\
& \qquad key(delete(l, (o, k)) \frown \langle e\rangle(i)) = k \\
\equiv\ & \{\text{divide into two cases.}\} \\
& \neg \exists\, i : \mathrm{dom}\, delete(l, (o, k)) \frown \langle e\rangle \bullet \\
& \qquad (i < \#(delete(l, (o, k)) \frown \langle e\rangle) \wedge \\
& \qquad\qquad op(delete(l, (o, k)) \frown \langle e\rangle(i)) = o \wedge \\
& \qquad\qquad key(delete(l, (o, k)) \frown \langle e\rangle(i)) = k) \vee \\
& \qquad (i = \#(delete(l, (o, k)) \frown \langle e\rangle) \wedge \\
& \qquad\qquad op(delete(l, (o, k)) \frown \langle e\rangle(i)) = o \wedge \\
& \qquad\qquad key(delete(l, (o, k)) \frown \langle e\rangle(i)) = k) && (4.4)
\end{aligned}$$

(4.4) is proven by contradiction. Suppose such $i : \mathrm{dom}\, delete(l, (o, k)) \frown \langle e\rangle$ exists. If $i = \#(delete(l, (o, k)) \frown \langle e\rangle)$, then

$$op(delete(l, (o, k)) \frown \langle e\rangle(i)) = op(e) = o$$

and

$$key(delete(l,(o,k)) \frown \langle e\rangle(i)) = key(e) = k$$

holds, which is contradictory to the assumption that $op(e) = o \wedge key(e) = k$ does NOT hold. If $i < \#(delete(l,(o,k)) \frown \langle e\rangle)$, then

$$op(delete(l,(o,k)) \frown \langle e\rangle(i)) = op(delete(l,(o,k))(i)) = o$$

and

$$key(delete(l,(o,k)) \frown \langle e\rangle(i)) = key(delete(l,(o,k))(i)) = k$$

holds, which is contradictory to I.H. Therefore, there is no such i. Thus, (4.4) has been proven by contradiction. Both the base case and the inductive step have been proven to be true. By the principle of structure induction on sequences, the lemma holds. □

Lemma 4.2.3 (No-time-shift of *delete*). *Given l : Log, o : OP, k : KEY, and t : Time, $isAfterLast(l,t) \Rightarrow isAfterLast(delete(l,(o,k)),t)$.*

Proof. Structure induction on l. **Base case.** $l = \langle\rangle$: Trivial. **Inductive step.** It should be proven that

$$isAfterLast(l \frown \langle e\rangle, t) \Rightarrow isAfterLast(delete(l \frown \langle e\rangle,(o,k)),t) \tag{4.5}$$

on the assumption that

$$isAfterLast(l,t) \Rightarrow isAfterLast(delete(l,(o,k)),t) \tag{I.H.}$$

holds, where $l \frown \langle e\rangle$: *Log* and e : *LogEntry*. If $op(e) = o \wedge key(e) = k$ holds,

$$\begin{aligned}
&\text{RHS of (4.5)}\\
\equiv\ &\{\text{Definition of } delete\}\\
&isAfterLast(delete(l,(o,k)),t)\\
\Leftarrow\ &\{\text{I.H.}\}\\
&isAfterLast(l,t)\\
\Leftarrow\ &\{\text{Definition of } isAfterLast \text{ and the assumption that } l \text{ is a log}\}\\
&isAfterLast(l \frown \langle e\rangle, t)\\
\equiv\ &\text{LHS of (4.5)}
\end{aligned}$$

If $op(e) = o \wedge key(e) = k$ does NOT hold,

$$\begin{aligned}
&\text{RHS of (4.5)}\\
\equiv\ &\{\text{Definition of } delete\}\\
&isAfterLast(delete(l,(o,k)) \frown \langle e\rangle, t)\\
\equiv\ &\{\text{Definition of } isAfterLast\}\\
&time(e) < t\\
\equiv\ &\{\text{Definition of } isAfterLast\}\\
&\text{LHS of (4.5)}
\end{aligned}$$

Since both the base case and the inductive step have been proven to be true, it follows by the principle of structure induction on sequences that the lemma holds. □

Lemma 4.2.4 (Order-preservation of *delete***).** *Given* $l : \text{seq}\, \mathit{LogEntry}$, $k : \mathit{KEY}$, *and* $o : \mathit{OP}$, (1) $l \in \mathit{Log} \Rightarrow \mathit{delete}(l, (o, k)) \in \mathit{Log}$. (2) $l \in \mathit{CELog} \Rightarrow \mathit{delete}(l, (o, k)) \in \mathit{CELog}$.

Proof. To prove (1), the following needs to be proven, by the definition of *Log*.

$$\mathit{isOrdered}(l) \Rightarrow \mathit{isOrdered}(\mathit{delete}(l, (o, k))). \tag{4.6}$$

This is done by using structure induction on l. **Base case.** $l = \langle\rangle$: Trivial. **Inductive step.** It needs to be proven that

$$\mathit{isOrdered}(l \frown \langle e \rangle) \Rightarrow \mathit{isOrdered}(\mathit{delete}(l \frown \langle e \rangle, (o, k))). \tag{4.7}$$

on the assumption that

$$\mathit{isOrdered}(l) \Rightarrow \mathit{isOrdered}(\mathit{delete}(l, (o, k))) \qquad \text{(I.H.)}$$

where $l \frown \langle e \rangle : \mathit{Log}$ and $e : \mathit{LogEntry}$. If $\mathit{op}(e) = o \wedge \mathit{key}(e) = k$ holds,

$$\begin{aligned}
&\text{RHS of (4.7)}\\
\equiv\ &\{\text{Definition of } \mathit{delete}\}\\
&\mathit{isOrdered}(\mathit{delete}(l, (o, k)))\\
\Leftarrow\ &\{\text{I.H.}\}\\
&\mathit{isOrdered}(l)\\
\Leftarrow\ &\{\text{Definition of } \mathit{isOrdered} \text{ and } \mathit{Log}\}\\
&\mathit{isOrdered}(l \frown \langle e \rangle)\\
\equiv\ &\text{LHS of (4.7)}
\end{aligned}$$

If $\mathit{op}(e) = o \wedge \mathit{key}(e) = k$ does NOT hold,

$$\begin{aligned}
&\text{RHS of (4.7)}\\
\equiv\ &\{\text{Definition of } \mathit{delete}\}\\
&\mathit{isOrdered}(\mathit{delete}(l, (o, k)) \frown \langle e \rangle)\\
\equiv\ &\{\text{Definition of } \mathit{isOrdered}\}\\
&\forall\, i, j : \text{dom}\, \mathit{delete}(l, (o, k)) \frown \langle e \rangle \bullet i < j \Rightarrow\\
&\qquad \mathit{time}(\mathit{delete}(l, (o, k)) \frown \langle e \rangle(i)) < \mathit{time}(\mathit{delete}(l, (o, k)) \frown \langle e \rangle(j))
\end{aligned}$$

Assuming that the LHS of (4.7) holds, $\mathit{isOrdered}(l)$ holds. According to (I.H.), $\mathit{isOrdered}(\mathit{delete}(l, (o, k)))$ holds. Thus, it is sufficient if it can be proven that

$$\begin{aligned}&\forall\, i,j : \operatorname{dom} delete(l,(o,k)) \frown \langle e\rangle \bullet\\ &\qquad i < j \land j = \#(delete(l,(o,k)) \frown \langle e\rangle) \Rightarrow \\ &\qquad\quad time(delete(l,(o,k)) \frown \langle e\rangle(i)) < time(delete(l,(o,k)) \frown \langle e\rangle(j))\end{aligned} \tag{4.8}$$

holds. (4.8) can be proven as follows.

$$\begin{aligned}&(4.8)\\ \Leftarrow\;& \forall\, i : \operatorname{dom} delete(l,(o,k)) \bullet time(delete(l,(o,k))(i)) < time(e)\\ \Leftarrow\;& \{\text{Lemma 4.2.1}\}\\ & \forall\, i : \operatorname{dom} l \bullet time(l(i)) < time(e)\\ \Leftarrow\;& \{\text{Considers a sequence containing } l \text{ only}\}\\ & \forall\, i : \operatorname{dom} l \frown \langle e\rangle \bullet i < \#(l \frown \langle e\rangle) \Rightarrow\\ &\qquad time(l \frown \langle e\rangle(i)) < time(l \frown \langle e\rangle(\#(l \frown \langle e\rangle)))\\ \Leftarrow\;& \{\text{let } j = \#(l \frown \langle e\rangle)\}\\ & \forall\, i,j : \operatorname{dom} l \frown \langle e\rangle \bullet i < j \Rightarrow time(l \frown \langle e\rangle(i)) < time(l \frown \langle e\rangle(j))\\ \equiv\;& \{\text{Definition of } isOrdered\}\\ & isOrdered(l \frown \langle e\rangle)\\ \equiv\;& \{\text{Definition of } Log\}\\ & l \frown \langle e\rangle \text{ is a log}\end{aligned}$$

Since both the base case and the inductive step have been proven to be true, (4.6) holds. This completes the proof for (1).

To prove (2), the following needs to be proven, by the definition of *CELog*.

$$\begin{aligned}&isOrdered(l) \land isCEL(l) \Rightarrow\\ &\qquad isOrdered(delete(l,(o,k))) \land isCEL(delete(l,(o,k)))\end{aligned} \tag{4.9}$$

Since (1) holds, it is sufficient if it can be proven that

$$isOrdered(l) \land isCEL(l) \Rightarrow isCEL(delete(l,(o,k))),$$

which can be proven by induction on l, as in (1). □

Theorem 4.2.1. *Given l : CELog and e : LogEntry, $concat(l,e)$ is still a characteristic-entry log.*

Proof. To prove $concat(l,e)$ is a characteristic-entry log, it needs to be proven that

1. $isOrdered(concat(l,e))$.
2. $isCEL(concat(l,e))$.

If $isAfterLast(l,e)$ does NOT hold, $concat(l,e) = l$. Thus (1) and (2) hold by the assumption that l is a characteristic-entry log. If $isAfterLast(l,e)$ holds, it needs to be proven that

$$isOrdered(delete(l, (op(e), key(e))) \frown \langle e \rangle) \tag{4.10}$$
$$isCEL(delete(l, (op(e), key(e))) \frown \langle e \rangle) \tag{4.11}$$

Through the assumption that l is a characteristic-entry log, it is known that both $isOrdered(l)$ and $isCEL(l)$ hold. Thus,

$$isOrdered(delete(l, (op(e), key(e))))$$
$$isCEL(delete(l, (op(e), key(e))))$$

hold by Lemma 4.2.4. To prove (4.10) and (4.11) it is hence sufficient if it can be proven that

$$\forall\, i : \operatorname{dom} delete(l, (op(e), key(e))) \bullet$$
$$\quad time(delete(l, (op(e), key(e)))(i)) < time(e) \tag{4.12}$$
$$\forall\, i : \operatorname{dom} delete(l, (op(e), key(e))) \bullet$$
$$\quad key(delete(l, (op(e), key(e)))(i)) = key(e) \Rightarrow$$
$$\qquad op(l(i)) \neq op(e) \tag{4.13}$$

By Lemma 4.2.3 and the assumption that $isAfterLast(l, e)$ holds, (4.12) holds. By Lemma 4.2.2, (4.13) holds. □

4.3 Converting normal logs into characteristic-entry logs

Given a log, characteristic entries can be filtered out to obtain its corresponding characteristic-entry log. The function ce is defined for this purpose.

$$ce : \operatorname{seq} LogEntry \to \operatorname{seq} LogEntry$$
$$ce = \lambda\, l : Log \bullet \textbf{if } l = \langle\rangle \textbf{ then } \langle\rangle \textbf{ else } concat(ce(front(l)), last(l))$$

For example, let l be a normal log, $l = \langle A;\ W;\ R;\ D;\ A;\ W \rangle$, recording a sequence of accesses. The characteristic-entry log of the same sequence of accesses can be obtained by applying the ce function to l, as follows.

$$\begin{aligned}
& ce(l) \\
= \ & concat(ce(front(l)), last(l)) \\
= \ & concat(ce(\langle A;\ W;\ R;\ D;\ A \rangle), W) \\
= \ & concat(concat(ce(\langle A;\ W;\ R;\ D \rangle), A), W) \\
= \ & concat(concat(concat(ce(\langle A;\ W;\ R \rangle), D), A), W) \\
= \ & concat(concat(concat(concat(ce(\langle A;\ W \rangle), R), D), A), W) \\
= \ & concat(concat(concat(concat(concat(ce(\langle A \rangle), W), R), D), A), W) \\
= \ & concat(concat(concat(concat(concat(concat(ce(\langle\rangle), A), W), R), D), A), W) \\
= \ & concat(concat(concat(concat(concat(concat(\langle\rangle, A), W), R), D), A), W) \\
= \ & concat(concat(concat(concat(concat(\langle A \rangle, W), R), D), A), W)
\end{aligned}$$

$$\begin{aligned}
&= concat(concat(concat(concat(\langle A;\ W\rangle, R), D), A), W)\\
&= concat(concat(concat(\langle A;\ W;\ R\rangle, D), A), W)\\
&= concat(concat(\langle A;\ W;\ R;\ D\rangle, A), W)\\
&= concat(\langle W;\ R;\ D;\ A\rangle, W)\\
&= \langle R;\ D;\ A;\ W\rangle
\end{aligned}$$

Theorem 4.3.1. *Given a log l, $ce(l)$ is a characteristic-entry log.*

Proof. The proof is obtained by applying structure induction to l. **Base case.** $l = \langle\rangle$: Trivial. **Inductive step.** It should be proven that $ce(l \frown \langle e\rangle)$ is a characteristic-entry log on the assumption that $ce(l)$ is a characteristic entry log and l and $l \frown \langle e\rangle$ are logs.

$$\begin{aligned}
&ce(l \frown \langle e\rangle)\\
=\ &\{\text{Definition of } ce\}\\
&concat(ce(l), e)
\end{aligned}$$

By the induction hypothesis, $ce(l)$ is a characteristic entry log. By Theorem 4.2.1, $concat(ce(l), e)$ is still a characteristic entry log. This establishes the inductive step. By structure induction, the theorem is proven. □

4.3.1 Properties of log conversion

Like the *delete* function, the *concat* and *ce* functions also have the property of non-creation.

Lemma 4.3.1 (Non-creation of *concat*). *Given $l \in Log$ and $e \in LogEntry$, $\forall i : \operatorname{dom} concat(l, e) \bullet \exists j : \operatorname{dom} l \frown e \bullet concat(l, e)(i) = l \frown \langle e\rangle(j)$.*

Proof. The proof of the lemma is divided into three cases. **(1)** If $l = \langle\rangle$, trivial. **(2)** If $l \neq \langle\rangle$ and $isAfterLast(l, e) = false$, trivial. **(3)** If $l \neq \langle\rangle$ and $isAfterLast(l, e) = true$,

$$\begin{aligned}
&\forall i : \operatorname{dom} concat(l, e) \bullet \exists j : \operatorname{dom} l \frown e \bullet concat(l, e)(i) = l \frown \langle e\rangle(j)\\
\equiv\ &\{\text{Definition of } concat,\ isAfterLast(l, e) = true\}\\
&\forall i : \operatorname{dom} delete(l, (op(l), key(e))) \frown \langle e\rangle \bullet\\
&\qquad \exists j : \operatorname{dom} l \frown e \bullet delete(l, (op(l), key(e))) \frown \langle e\rangle(i) = l \frown \langle e\rangle(j)
\end{aligned}$$

To prove that given any $i : \operatorname{dom} delete(l, (op(l), key(e))) \frown \langle e\rangle$, there exists $j : \operatorname{dom} l \frown e$ such that $delete(l, (op(l), key(e))) \frown \langle e\rangle(i) = l \frown \langle e\rangle(j)$, two situations will be considered.

- If $i < \#delete(l, (op(e), key(e))) \frown \langle e \rangle$, $i \leq \#delete(l, (op(e), key(e)))$ holds. According to Lemma 4.2.1, it is known that there exists $j : \mathrm{dom}\, l$ such that $delete(l, (op(e), key(e)))(i) = l(j)$. Thus,

$$\begin{aligned} & delete(l, (op(l), key(e))) \frown \langle e \rangle(i) \\ = & \; delete(l, (op(l), key(e)))(i) \\ = & \; l(j) \\ = & \; l \frown \langle e \rangle(j) \end{aligned}$$

- If $i = \#delete(l, (op(e), key(e))) \frown \langle e \rangle$, simply let $j = \#(l) + 1$. Thus,

$$\begin{aligned} & delete(l, (op(l), key(e))) \frown \langle e \rangle(i) \\ = & \; e \\ = & \; l \frown \langle e \rangle(j) \end{aligned}$$

□

Lemma 4.3.2 (Non-creation of *ce*). *Given $l : Log$, $\forall\, i : \mathrm{dom}\, ce(l) \bullet \exists\, j : \mathrm{dom}\, l \bullet ce(l)(i) = l(j)$.*

Proof. Structure induction on l. **Base case.** $l = \langle\rangle$: Trivial. **Inductive step.** It should be proven that

$$\forall\, i : \mathrm{dom}\, ce(l \frown \langle e \rangle) \bullet \exists\, j : \mathrm{dom}\, l \frown \langle e \rangle \bullet ce(l \frown \langle e \rangle)(i) = l \frown \langle e \rangle(j) \tag{4.14}$$

on the assumption that

$$\forall\, i : \mathrm{dom}\, ce(l) \bullet \exists\, j : \mathrm{dom}\, l \bullet ce(l)(i) = l(j)$$

$$\begin{aligned} & (4.14) \\ \equiv & \; \{\text{Definition of } ce\} \\ & \forall\, i : \mathrm{dom}\, concat(ce(l), e) \bullet \\ & \qquad \exists\, j : \mathrm{dom}\, l \frown \langle e \rangle \bullet concat(ce(l), e)(i) = l \frown \langle e \rangle(j) \end{aligned}$$

It should be proven that given any $i : \mathrm{dom}\, concat(ce(l), e)$, there exists $j : \mathrm{dom}\, l \frown \langle e \rangle$ such that $concat(ce(l), e)(i) = l \frown \langle e \rangle(j)$. According to Lemma 4.3.1, it is known that there exists $j^{'} : \mathrm{dom}\, ce(l) \frown \langle e \rangle$ such that $concat(ce(l), e)(i) = ce(l) \frown \langle e \rangle(j')$. Now, two cases will be considered.

- If $j' < \#(ce(l) \frown \langle e \rangle)$, then $j' \leq ce(l)$. By the induction hypothesis, there exists $j : \mathrm{dom}\, l$ such that $ce(l)(j') = l(j)$. Thus,

$$\begin{aligned} & concat(ce(l), e)(i) \\ = & \; ce(l) \frown \langle e \rangle(j') \\ = & \; ce(l)(j') \\ = & \; l(j) \end{aligned}$$

- If $j' = \#(ce(l) \frown \langle e \rangle)$, simply let $j = \#(l \frown \langle e \rangle)$. Thus,

$$\begin{aligned} & concat(ce(l), e)(i) \\ = {} & ce(l) \frown \langle e \rangle (j') \\ = {} & e \\ = {} & l \frown \langle e \rangle (j) \end{aligned}$$

□

The result of the *ce* function also preserves the timing of the original log, as shown by the following two lemmas.

Lemma 4.3.3 (Time-preservation of *ce*). *Given $l : Log$,*

$$isAfterLast(ce(front(l)), time(last(l))) \tag{4.15}$$

Proof. Two cases will be considered. **(1)** If $ce(front(l)) = \langle\rangle$, (4.15) holds by the definition of *isAfterLast*. **(2)** If $ce(front(l)) \neq \langle\rangle$, let $i = \#(ce(front(l)))$. There exists $j : front(l)$ such that $ce(front(l))(i) = front(l)(j)$, by Lemma 4.3.2. Therefore,

$$\begin{aligned} & (4.15) \\ \equiv {} & \{\text{Definition of } isAfterLast,\ ce(front(l)) \neq \langle\rangle\ \} \\ & time(last(ce(front(l)))) < time(last(l)) \\ \equiv {} & \{\text{Definition of } last,\ i = \#(ce(front(l)))\} \\ & time(ce(front(l))(i)) < time(last(l)) \\ \equiv {} & \{\text{Lemma 4.3.2}\} \\ & time(front(l)(j)) < time(last(l)) \\ \equiv {} & \{\text{Definition of } front,\ j \leq \#(front(l)) < \#(l)\} \\ & time(l(j)) < time(last(l)) \\ \equiv {} & \{j \leq \#(front(l)) < \#(l),\ l \text{ is a log}\} \\ & true \end{aligned}$$

Thus, it is proven that (4.15) holds in both cases. □

Lemma 4.3.4 (Latest-equivalence of *ce*). *Given $l \in Log$ and $t \in Time$, $isAfterLast(l, t) \equiv isAfterLast(ce(l), t)$.*

Proof. Let LHS = $isAfterLast(l, t)$ and RHS = $isAfterLast(ce(l), t)$. The proof of the lemma is divided into two cases. **(1)** If $l = \langle\rangle$, trivial. **(2)** If $l \neq \langle\rangle$,

$$\begin{aligned} & \text{RHS} \\ \equiv {} & \{\text{Definition of } ce\} \\ & isAfterLast(concat(ce(front(l)), last(l)), t) \\ \equiv {} & \{\text{Definition of } concat, \text{ Lemma 4.3.3}\} \end{aligned}$$

$$\begin{tikzcd}
\text{Log} \arrow[r, "\text{ce}"] \arrow[d, "\text{append(., e)}"'] & \text{CELog} \arrow[d, "\text{concat(.,e)}"] \\
\text{Log} \arrow[r, "\text{ce}"] & \text{CELog}
\end{tikzcd}$$

Figure 4.2. The relations between logs and characteristic-entry logs.

$$\begin{aligned}
& \mathit{isAfterLast}(\mathit{delete}(\mathit{ce}(\mathit{front}(l)), (\mathit{op}(\mathit{last}(l)), \mathit{key}(\mathit{last}(l)))) \frown \langle \mathit{last}(l) \rangle, t) \\
\equiv\ & \{\text{Definition of } \mathit{isAfterLast}\} \\
& \mathit{time}(\mathit{last}(l)) < \mathit{time}(t) \\
\equiv\ & \text{LHS}
\end{aligned}$$

Thus, it is proven that the lemma holds in both cases. □

The commutability theorem shows that a characteristic-entry log can be constructed on the fly and so the long normal logs can be replaced by the compact characteristic-entry logs, as illustrated in Figure 4.2.

Theorem 4.3.2 (Commutability). *Given any log, $l \in \mathit{Log}$, and any log entry, $e \in \mathit{LogEntry}$, $\mathit{ce}(\mathit{append}(l, e)) = \mathit{concat}(\mathit{ce}(l), e)$.*

Proof. Let LHS = $\mathit{ce}(\mathit{append}(l, e))$ and RHS = $\mathit{concat}(\mathit{ce}(l), e)$. The proof of the theorem is divided into three cases. **(1)** If $l = \langle\rangle$, trivial. **(2)** If $l \neq \langle\rangle$ and $\mathit{isAfterLast}(l, \mathit{time}(e)) = \mathit{true}$, $\mathit{isAfterLast}((\mathit{ce}(l), \mathit{time}(e))$ also holds by Lemma 4.3.4.

$$\begin{aligned}
& \text{LHS} \\
\equiv\ & \{\mathit{isAfterLast}(l, \mathit{time}(e)) = \mathit{true}, \text{ Definition of } \mathit{append}\} \\
& \mathit{ce}(l \frown \langle e \rangle) \\
\equiv\ & \{\text{Definition of } \mathit{ce}\} \\
& \mathit{concat}(\mathit{ce}(l), e) \\
\equiv\ & \text{RHS}
\end{aligned}$$

Therefore, it is proven that

$$\mathit{isAfterLast}(l, \mathit{time}(e)) = \mathit{true} \Rightarrow \mathit{ce}(\mathit{append}(l, e)) = \mathit{concat}(\mathit{ce}(l), e).$$

(3) If $l \neq \langle\rangle$ and $\mathit{isAfterLast}(l, \mathit{time}(e)) = \mathit{false}$, $\mathit{isAfterLast}((\mathit{ce}(l), \mathit{time}(e))$ does not hold either by Lemma 4.3.4.

$$\begin{aligned}
& \text{LHS} \\
\equiv\ & \{\mathit{isAfterLast}(l, \mathit{time}(e)) = \mathit{false}, \text{ Definition of } \mathit{append}\ \}
\end{aligned}$$

$$\begin{aligned}
& ce(l) \\
\equiv\ & \{ isAfterLast(ce(l), time(e) = false), \text{Definition of } concat\ \} \\
& \text{RHS}
\end{aligned}$$

Thus, it is proven that

$$isAfterLast(l, time(e)) = false \Rightarrow ce(append(l, e)) = concat(ce(l), e)$$

The theorem holds for all values of l. □

The proof of the commutability of *ce* is actually not a "deep" rewriting process. It follows easily because of the way *ce* is defined recursively.

4.4 Valid characteristic-entry logs

The execution of certain operations on data spaces must comply with certain constraint requirements, as shown in Section 3.2. For instance, adding a data item to a data space will succeed only if the data space does not contain the key of the data item. Constraint requirements result in a collection of valid logs, which were studied in Section 3.3.1. Constraint requirements concerning executions of operations also determine what a characteristic-entry log looks like.

Table 4.1 shows all the possible instances of a characteristic-entry log with respect to the validity expressed in Section 3.3.1. For example, ϵ is a valid instance of a characteristic-entry log, standing for no operation performed on the given key. D is not a valid instance, since the key of the given data item is not defined in the given data space. A; D is a valid instance of a characteristic-entry log, because it is the result of any times of iteration of add and delete operations.

Theorem 4.4.1 (Validity of Characteristic-entry logs). *A valid characteristic entry log of a key has one of the following forms:* $\langle\rangle$, $\langle A\rangle$, $\langle A;\ D\rangle$, $\langle A;\ W\rangle$, $\langle D;\ A\rangle$, $\langle A;\ W;\ R\rangle$, $\langle A;\ R;\ W\rangle$, $\langle D;\ A;\ W\rangle$, $\langle A;\ W;\ D\rangle$, $\langle W;\ A;\ D\rangle$, $\langle W;\ D;\ A\rangle$, $\langle D;\ A;\ W;\ R\rangle$, $\langle D;\ A;\ R;\ W\rangle$, $\langle R;\ D;\ A;\ W\rangle$, $\langle A;\ W;\ R;\ D\rangle$, $\langle A;\ R;\ W;\ D\rangle$, $\langle R;\ A;\ W;\ D\rangle$, $\langle W;\ R;\ A;\ D\rangle$, $\langle R;\ W;\ A;\ D\rangle$, $\langle W;\ R;\ D;\ A\rangle$, $\langle R;\ W;\ D;\ A\rangle$.

Proof. The proof of this theorem is as follows. First, find all combinations of four operations. Then verify each combination against the validity. Table 4.1 illustrates this process. Table 4.2 lists all the valid characteristic-entry logs. For each valid characteristic-entry log, an example of a valid log instance is given. □

With an enumeration of all possible appearances of a valid characteristic-entry log, it is easy to define the predicate telling whether a given instance

Table 4.1. Possible sequences of operations on a key in a characteristic-entry log. Assume that all the operations in a sequence are related to a key, say k. The key is omitted in sequences, for simplicity. In a sequence, operations are ordered by time. The right-most operation is the last performed. An X in a column other than the first one means that the actual log instance violates the property indicated above of the column. An X in the last column simply means that the instance is not valid.

Instance	Declaration	Initialization	Deletion	Addition	Validity
ϵ					
A					
D	X				X
W	X				X
R	X				X
A; D					
A; W					
A; R		X			X
D; A					
D; R	X		X		X
D; W	X		X		X
R; A	X				X
R; D	X				X
R; W	X				X
W; A					X
W; D	X				X
W; R	X				X
D; W; R	X		X		X
A; W; R					
W; D; R	X		X		X
A; D; R		X	X		X
W; A; R	X	X			X
D; A; R		X			X
D; R; W	X		X		X
A; R; W					
R; D; W	X		X		X
A; D; W			X		X
R; A; W	X				X
D; A; W					
W; R; D	X				X
A; R; D		X			X
R; W; D	X				X
A; W; D					
R; A; D	X				X
W; A; D					
W; R; A	X				X
D; R; A			X		X
R; W; A	X				X
D; W; A			X		X
R; D; A		X			X
W; D; A					
A; D; W; R			X		X
D; A; W; R					
A; W; D; R			X		X
W; A; D; R		X	X		X

Table 4.1 Continued.

Instance	Declaration	Initialization	Deletion	Addition	Validity
$D;\ W;\ A;\ R$		X	X		X
$W;\ D;\ A;\ R$		X			X
$A;\ D;\ R;\ W$			X		X
$D;\ A;\ R;\ W$					
$A;\ R;\ D;\ W$			X		X
$R;\ A;\ D;\ W$			X		X
$D;\ R;\ A;\ W$			X		X
$R;\ D;\ A;\ W$					
$A;\ W;\ R;\ D$					
$W;\ A;\ R;\ D$		X			X
$A;\ R;\ W;\ D$					
$R;\ A;\ W;\ D$					
$W;\ R;\ A;\ D$					
$R;\ W;\ A;\ D$					
$D;\ W;\ R;\ A$			X		X
$W;\ D;\ R;\ A$			X		X
$D;\ R;\ W;\ A$			X		X
$R;\ D;\ W;\ A$			X		X
$W;\ R;\ D;\ A$					
$R;\ W;\ D;\ A$					

$$\begin{tikzcd}
\text{Log} \arrow[r, "\text{ce}"] \arrow[d, "\text{isValidLog(.)}"'] & \text{CELog} \arrow[d, "\text{isValidCELog(.)}"] \\
\text{ValidLog} \arrow[r, "\text{ce}"] & \text{ValidCELog}
\end{tikzcd}$$

Figure 4.3. The relations between valid logs and valid characteristic-entry logs.

of a characteristic-entry log is valid or not. A valid characteristic-entry log should satisfy the property defined by the predicate *isValidCELog*.

$$\begin{array}{|l}
\mathit{isValidCELog}\_ : \mathbb{P}(\mathrm{seq}\,\mathit{LogEntry}) \\
\hline
\forall\, l : \mathrm{seq}\,\mathit{LogEntry} \bullet \mathit{isValidCELog}(l) \Leftrightarrow \\
\quad \forall\, k : \mathit{KEY} \bullet \mathit{ops}(l,k) \in \{\langle\rangle, \langle A\rangle, \langle A;\ D\rangle, \langle A;\ W\rangle, \langle D;\ A\rangle, \\
\quad \langle A;\ W;\ R\rangle, \langle A;\ R;\ W\rangle, \langle D;\ A;\ W\rangle, \langle A;\ W;\ D\rangle, \langle W;\ A;\ D\rangle, \\
\quad \langle W;\ D;\ A\rangle, \langle D;\ A;\ W;\ R\rangle, \langle D;\ A;\ R;\ W\rangle, \langle R;\ D;\ A;\ W\rangle, \\
\quad \langle A;\ W;\ R;\ D\rangle, \langle A;\ R;\ W;\ D\rangle, \langle R;\ A;\ W;\ D\rangle, \langle W;\ R;\ A;\ D\rangle, \\
\quad \langle R;\ W;\ A;\ D\rangle, \langle W;\ R;\ D;\ A\rangle, \langle R;\ W;\ D;\ A\rangle\}
\end{array}$$

Thus, validity requirements of logs can be imposed on characteristic-entry logs, which is illustrated by the commuting diagram in Figure 4.3.

Table 4.2. Example instances of valid characteristic-entry logs.

Valid characteristic-entry logs	instances of valid logs
ϵ	$\langle\rangle$
A	A
$A;\ D$	$A;\ D$
$A;\ W$	$A;\ W$
$D;\ A$	$A;\ D;\ A$
$A;\ W;\ R$	$A;\ W;\ R$
$A;\ R;\ W$	$A;\ W;\ R;\ W$
$D;\ A;\ W$	$A;\ D;\ A;\ W$
$A;\ W;\ D$	$A;\ W;\ D$
$W;\ A;\ D$	$A;\ W;\ D;\ A;\ D$
$W;\ D;\ A$	$A;\ W;\ D;\ A$
$D;\ A;\ W;\ R$	$A;\ D;\ A;\ W;\ R$
$D;\ A;\ R;\ W$	$A;\ D;\ A;\ W;\ R;\ W$
$R;\ D;\ A;\ W$	$A;\ W;\ R;\ D;\ A;\ W$
$A;\ W;\ R;\ D$	$A;\ W;\ R;\ D$
$A;\ R;\ W;\ D$	$A;\ W;\ R;\ W;\ D$
$R;\ A;\ W;\ D$	$A;\ W;\ R;\ D;\ A;\ W;\ D$
$W;\ R;\ A;\ D$	$A;\ W;\ R;\ D;\ A;\ D$
$R;\ W;\ A;\ D$	$A;\ W;\ R;\ W;\ D;\ A;\ D$
$W;\ R;\ D;\ A$	$A;\ W;\ R;\ D;\ A$
$R;\ W;\ D;\ A$	$A;\ W;\ R;\ W;\ D;\ A$

The full specification of the valid characteristic-entry logs is obtained by combining the element specifications developed above.

$$ValidCELog == \{l : CELog \mid isValidCELog(l)\}.$$

Given a key, all valid instances of a characteristic-entry log entries of the key can be computed. Suppose the execution constraints of operations are as defined in Figure 3.3.1. All valid characteristic operation sequences can be calculated using the following algorithm.

Algorithm. Let S_{ce} denote all valid characteristic operation sequences. More precisely, each element of S_{ce} has the form (s, l), where s is a state of FSM_c and l is a characteristic operation sequence. Initially, S_{ce} is empty.

1. Add $(0, \epsilon)$ to S_{ce}.
2. For each $(s, l) \in S_{ce}$, and $o \in OP$, if $move(s, o)$ is a valid move in FSM_c and $ce(l.o) \notin S_{ce}$, add $(move(s, o), ce(l.o))$ to S_{ce}.
3. $\{l \mid (s, l) \in S_{ce}\}$ contains all the valid characteristic operation sequences.

Figure 4.4 illustrates how the algorithm works with respect to the validity expressed in Figure 3.1. In the figure, a finite state machine is used to show intermediate steps. Every state in the figure is indicated by $s : l$, representing (s, l) in the algorithm.

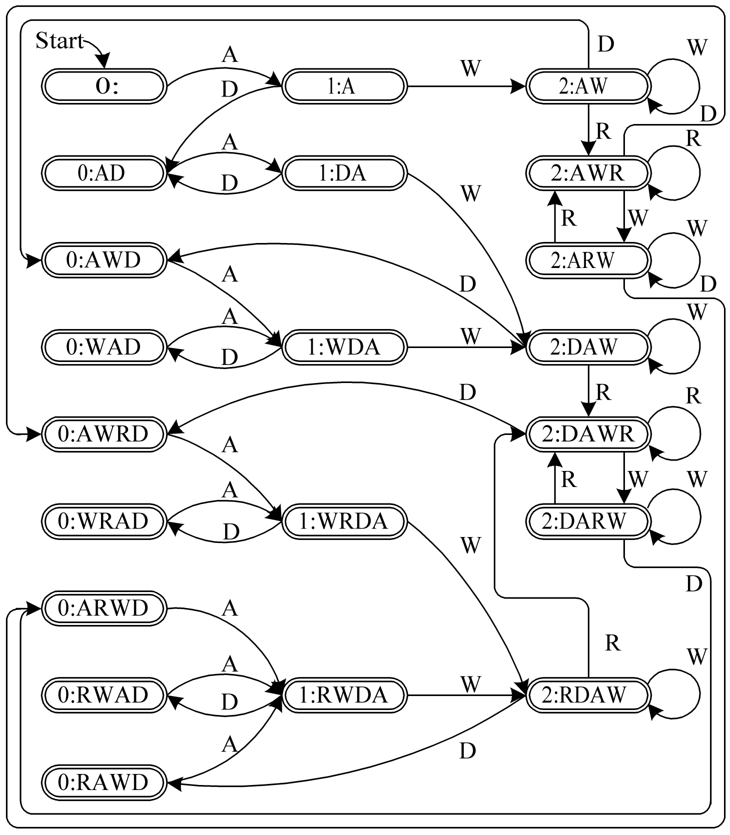


Figure 4.4. Each state in this finite-state machine contains a valid form of a characteristic-entry log.

4.5 Concluding remarks

In this chapter, characteristic-entry logs have been formally studied. First, characteristic-entry logs were modelled as mathematical structures, notable sequences, in the Z notation. Next, a mathematical function for constructing characteristic-entry logs was defined. Related properties of the function were presented and proven. After that, a conversion function for converting normal logs to characteristic-entry logs was defined. Finally, it was investigated how validity requirements relating to the execution of data accesses can be formulated in terms of characteristic-entry logs. The formal specification of characteristic-entry logs can be used as a reference implementation in testing.

Formal methods have been applied in a few data synchronization systems. Predicate calculus is used to specify the behavior of Unison in [10, 113]. In

Unison, a file system is modelled as a mapping from filenames to data objects and synchronization is considered to be fully state-based. Unison stores only archives (states) between successful synchronizations and does not record operations on files at all. It does not model access histories. Filesystem algebra [125] gives an algebraic specification of file synchronization. The algebraic approach derives from the work of Unison. It does not model operation logs.

Distributed systems take advantage of access histories recorded in log files to resolve data inconsistencies. In the Coda system [84], resolution logs keep track of adding, removing, and updating activities on replicated directories. In the Ficus replicated file system [126, 51], each file replica has its own version vector that records the history of updates on the file [109]. In database systems, system states can be rolled back, after which uncommitted transactions recorded in logs can be serialized and applied to all data copies [29]. In those systems, limits of log sizes, derived from empirical data, are often determined by system administrators. Various log truncation methods are proposed and several semantic rules are applied in data synchronization without rigorous definitions and proofs. Few of the above-mentioned existing systems applied formal specification techniques in designing log mechanisms.

5. Using characteristic-entry logs in data synchronization

Characteristic-entry logs can be used in semantic data synchronization in the same way as normal logs. In this chapter, several semantic rules will be introduced first. Next, each rule will be formally specified. Finally, it will be rigorously proven that characteristic-entry logs contain sufficient information for semantic data synchronization, for the introduced rules.

5.1 Semantic rules

Generally speaking, there are two ways of using logs in data synchronization.

- Serialization. In this approach, systems are rolled back to an initial consistent state. Then entries of logs of data spaces are serialized according to their timestamps.[1] The serialized log entries are applied to all data spaces, after which the system state becomes consistent.
- Semantic synchronization. In this approach, semantic rules are defined to resolve inconsistencies, such as "propagating the latest update".

Serialization makes full use of the information that is stored in logs. In this approach, however, rolling back system states and applying serialized logs are expensive operations in terms of computing resources, especially to mobile devices. Semantic synchronization incorporates user knowledge in data synchronization and provides flexibility of defining consistency. In this approach, however, only a small portion of the information stored in logs is used. There is much redundancy in logs. Characteristic-entry logs can be used in the same way as normal logs in semantic data synchronization while avoiding redundancy.

[1] In theory it is impossible to keep system clocks synchronized. In practice, algorithms such as the Network Time Protocol (NTP) [99] can ensure computer clocks to be synchronized to within a few tens of milliseconds in today's global Internet. The improved NTP [100] achieves an accuracy within 10 milliseconds on typical Internet paths. For most file sharing applications in home environments such accuracies will be sufficient. If two log entries do happen to have the same timestamp, a time-collision error will be reported to users for manual repair.

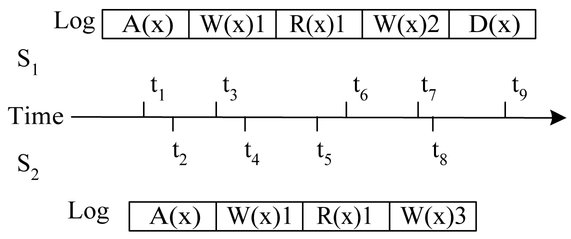


Figure 5.1. Update loss when applying the up-to-date rule.

In semantic data synchronization, the up-to-date rule is often used for its simplicity, especially in file synchronization tools.

Definition 5.1.1 (Up-to-date). *When a data item and its copies are synchronized, the most recent operation (modification) will be propagated to all the copies.*

Figure 5.1 illustrates this rule by an example. The most recent operation on x at S_2 was $W(x)3$. After that, there was a $D(x)$ operation at S_1. According to the rule, the deletion is chosen for propagation. Thus x is deleted from both S_1 and S_2 after synchronization.

Using the up-to-date rule carelessly might result in unexpected data loss. Imagine the following scenario. A user modified a file at one computer. After that, he deleted a copy of the file at another computer, just in order to clean up local storage space. When the two computers are synchronized, the file will be removed from both computers according to the up-to-date rule, since the deletion was performed most recently. This is not the user's initial intention. To avoid such data loss, Ficus introduced "no lost update" semantics [126]. This rule is stated as follows.

Definition 5.1.2 (No-update-loss). *When a data item and its copies are synchronized, the most recent modification will be propagated to all copies.*

Note that a "modification" here means updating the content of a file. Removing a file is not considered to be a modification. Take the example in Figure 5.1. $W(x)3$ is chosen for propagation in synchronization, according to the no-update-loss rule. In this way, the latest modification made on S_2 becomes available on S_1, even though x was already deleted.

The no-update-loss rule prevents the risk of modifications being lost by a delete operation, which might have the side-effect that a delete operation always "deletes nothing" after the synchronization. Figure 5.2 illustrates an example. In this scenario, $D(x)$ is ignored and $W(x)2$ is chosen for propagation, according to the no-update-loss rule. Note that in this example, both $W(x)2$ and $D(x)$ in S_1 occurred after $W(x)3$ in S_2, indicating the most recent operations were performed on S_1. The user might want to apply all the changes made on S_1 to S_2. Using the no-update-loss rule will not propagate

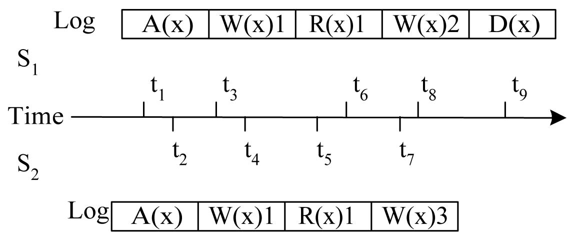


Figure 5.2. The delete operation is discarded when applying the no-update-loss rule.

the deletion at all. To enable the deletion without losing the safety aspect of the no-update-loss rule, the weak-no-update-loss rule is introduced.

Definition 5.1.3 (Weak-no-update-loss). *When a data item and its copies are synchronized, the most recent modification will be propagated to all copies if this modification has not yet been deleted; otherwise, the most recent operation will be propagated.*

According to this rule, $W(x)3$ is chosen for propagation in the example of Figure 5.1. In the example of Figure 5.2, $D(x)$ is chosen for propagation.

More subtle rules are used in semantic data synchronization, such as *read-delete-safety* in [122]. In the subsequent sections it will be rigorously proven that when the above-mentioned rules are applied in semantic data synchronization, characteristic-entry logs can be used in the same way as normal logs. To this end, semantic rules must be formalized.

Remark. In professional database applications, such as banking systems and flight reservation systems, dependence between updates is of importance. For example, concurrent transactions of money transfer should be scheduled in such a way that at any moment in the execution of the transactions, the amount of money being transferred will always be less than the credit of the account. In this example, the same information item is accessed in a short time period by several clients.

In the context of the Phenom project, updates usually mean modifications on descriptive information (annotation) of personal digital photos, such as image contents or the location where images were taken. Digital images are personal information, which is less likely to be accessed and updated by several persons at the same time than working documents in office environments. Furthermore, digital photos are not often updated in home environments after they have been taken. Thus, it is realistic to regard updates on photo-like digital media data objects as atomic and isolated events. The dependence among updates is not taken into account in the model. For the same reason, any dependency among data items is not considered either.

5.2 Formalizing semantic rules

To prove the correctness of using characteristic-entry logs in semantic data synchronization, the semantic rules will be formalized. Some conventions will be introduced here.

In a DU system it is assumed that there is no data dependency between data items. Typical examples of data items in such a system are files. A file is usually accessed independently of other files. Moreover, in a DU system, data accesses are regarded as atomic and isolated events. By leaving access dependencies out of the model, a better understanding of semantic rules can be obtained. In the rest of the chapter one-key logs, whose entries are associated with the same key, will therefore be used as research carriers.

$$\mathit{OneKeyLog} == \{l : \mathit{Log} \mid \mathit{onekey}(l)\}$$

where

$$\begin{array}{|l}
\mathit{onekey}\_ : \mathbb{P}(\mathrm{seq}\,\mathit{LogEntry}) \\
\hline
\forall\, l : \mathrm{seq}\,\mathit{LogEntry} \bullet \mathit{onekey}(l) \Leftrightarrow \forall\, i, j : \mathrm{dom}\, l \bullet \mathit{key}(l(i)) = \mathit{key}(l(j))
\end{array}$$

Semantic rules can be modelled as partial functions. The following abbreviation is introduced for future use.

$$\mathit{RULE} == \mathit{OneKeyLog} \times \mathit{OneKeyLog} \mathrel{+\!\!\!\rightarrow} \mathrm{seq}\,\mathit{LogEntry} \times \mathrm{seq}\,\mathit{LogEntry}.$$

A semantic rule takes a pair of logs and computes what operations should be chosen for propagation. The results are captured in a pair of sequences of log entries, which should be applied to the data spaces with which the logs are associated, in a component-wise manner. A semantic rule is *totally defined* if it can be applied to any pairs of logs.

Moreover, a new date type OP_ϵ is introduced, which is an extension of the basic data type OP. OP_ϵ has an extra element ϵ, denoting an empty operation.

$$OP_\epsilon ::= OP \cup \{\epsilon\}.$$

The function last_ϵ is introduced, which returns the last operation of a sequence of log entries.

$$\begin{array}{|l}
\mathit{last}_\epsilon : \mathrm{seq}\,\mathit{LogEntry} \rightarrow OP_\epsilon \\
\hline
\mathit{last}_\epsilon = \lambda\, l : \mathrm{seq}\,\mathit{LogEntry} \bullet \textbf{if } l = \langle\rangle \textbf{ then } \epsilon \textbf{ else } \mathit{op}(\mathit{last}(l))
\end{array}$$

When specifying a semantic rule, the operation of the most recent access is often needed. To obtain such information, using $\mathit{op}(\mathit{last}(l))$ would cause undefinedness, where l is a sequence of log entries. This is because last is a partial function on sequences in Z. The introduction of the empty operation ϵ and the last_ϵ function helps to simplify specifications, as will be illustrated below in the definition of the domain of $\mathit{up}_{\neg\epsilon}$.

5.2.1 Up-to-date

The up-to-date rule can be formalized as the following function.

```
$up2date : RULE$
---
$up2date = \lambda\, l_1, l_2 : OneKeyLog \bullet$
    **if** $last_\epsilon(l_1) \neq \epsilon \wedge last_\epsilon(l_2) \neq \epsilon$
    **then if** $time(last(l_1)) < time(last(l_2))$ **then** $(\langle last(l_2)\rangle, \langle\rangle)$
        **else if** $time(last(l_2)) < time(last(l_1))$ **then** $(\langle\rangle, \langle last(l_1)\rangle)$
            **else** $(\langle\rangle, \langle\rangle)$
    **else if** $last_\epsilon(l_1) \neq \epsilon$ **then** $(\langle\rangle, \langle last(l_1)\rangle)$
        **else if** $last_\epsilon(l_2) \neq \epsilon$ **then** $(\langle last(l_2)\rangle, \langle\rangle)$
            **else** $(\langle\rangle, \langle\rangle)$
```

For example, let l_1 be the log of S_1 and l_2 be the log of S_2 in Figure 5.1. Note that $time(last(l_1)) = t_9$, $time(last(l_2)) = t_8$, and $t_9 > t_8$, where $last(l_1) = D(x)$ and $last(l_2) = W(x)3$.

$$up2date(l_1, l_2) = (\langle\rangle, \langle D(x)\rangle).$$

Thus, $D(x)$ should be applied to S_2 while nothing needs to be applied to S_1.

If this approach were to be used to describe sophisticated rules, the specification might become incomprehensible. A compositional approach will therefore be devised. A semantic rule is decomposed into small segments, each of which is modelled as a partial function separately. A full specification of the given semantic rule can be obtained by composing the individually specified segments. The full specification should be totally defined.

The up-to-date rule can be redefined in the compositional manner as follows. $up_{\neg\epsilon}$ is a partial function, which takes pairs of non-ϵ logs as arguments. The domain of $up_{\neg\epsilon}$ is restricted by the "dom" function. up_ϵ is a partial function, whose arguments involve at least one empty log.

```
$up_{\neg\epsilon} : RULE$
---
dom $up_{\neg\epsilon} = \{(l_1, l_2) : OneKeyLog \times OneKeyLog \mid \neg endwithR(l_1, l_2) \wedge$
    $last_\epsilon(l_1) \neq \epsilon \wedge last_\epsilon(l_2) \neq \epsilon\}$
$up_{\neg\epsilon} = \lambda\, l_1, l_2 : OneKeyLog \bullet$
    **if** $time(last(l_1)) < time(last(l_2))$ **then** $(\langle last(l_2)\rangle, \langle\rangle)$
    **else if** $time(last(l_2)) < time(last(l_1))$ **then** $(\langle\rangle, \langle last(l_1)\rangle)$
        **else** $(\langle\rangle, \langle\rangle)$
```

$$
\begin{array}{|l}
up_\epsilon : RULE \\
\hline
\mathrm{dom}\, up_\epsilon = \{(l_1, l_2) : OneKeyLog \times OneKeyLog \mid \neg endwithR(l_1, l_2) \wedge \\
\quad (last_\epsilon(l_1) = \epsilon \vee last_\epsilon(l_2) = \epsilon)\} \\
up_\epsilon = \lambda\, l_1, l_2 : OneKeyLog \bullet \\
\quad \textbf{if } last_\epsilon(l_1) \neq \epsilon \textbf{ then } (\langle\rangle, \langle last(l_1)\rangle) \\
\quad \textbf{else if } last_\epsilon(l_2) \neq \epsilon \textbf{ then } (\langle last(l_2)\rangle, \langle\rangle) \\
\qquad \textbf{else } (\langle\rangle, \langle\rangle)
\end{array}
$$

In the definitions, the predicate *endwithR* verifies whether either of two given logs ends with a read access.

$$
\begin{array}{|l}
endwithR\_ : \mathbb{P}(OneKeyLog \times OneKeyLog) \\
\hline
\forall\, l_1, l_2 : OneKeyLog \bullet endwithR(l_1, l_2) \Leftrightarrow \\
\quad (last_\epsilon(l_1) = read \vee last_\epsilon(l_2) = read)
\end{array}
$$

In semantic data synchronization, read accesses are usually not taken into account, which is modelled by the up_r function.[2] The up_r function takes two logs as arguments. If either of the logs ends with a read access, the access is ignored and the rest of the log is used in synchronization. Thus, the full specification of the up-to-date rule is the integration of up_ϵ, $up_{\neg\epsilon}$ and up_r.[3]

$$up \mathrel{\widehat{=}} up_{\neg\epsilon} \oplus up_\epsilon \oplus up_r$$

where

$$
\begin{array}{|l}
up_r : RULE \\
\hline
\mathrm{dom}\, up_r = \{(l_1, l_2) : OneKeyLog \times OneKeyLog \mid endwithR(l_1, l_2)\} \\
up_r = \lambda\, l_1, l_2 : OneKeyLog \bullet \\
\quad \textbf{if } last_\epsilon(l_1) = read \wedge last_\epsilon(l_2) = read \textbf{ then } up(front(l_1), front(l_2)) \\
\quad \textbf{else if } last_\epsilon(l_1) = read \textbf{ then } up(front(l_1), l_2) \\
\qquad \textbf{else } up(l_1, front(l_2))
\end{array}
$$

To find out how to use *up* in data synchronization, take the example in Figure 5.1. Let l_1 and l_2 be the logs of the data spaces S_1 and S_2 in Figure 5.1, respectively.

$$
\begin{aligned}
l_1 &= \langle A(x)_1,\, W(x)1_1, R(x)1_1,\, W(x)2_1, D(x)_1\rangle \\
l_2 &= \langle A(x)_2,\, W(x)1_2, R(x)1_2,\, W(x)3_2\rangle
\end{aligned}
$$

Synchronization can be effected by applying the *up* function to l_1 and l_2.

[2] It might be tempting to think that all read accesses are irrelevant and should perhaps not be included in logs at all. In [122] it is shown that read accesses can be used by applications in refining semantic rules.

[3] The overriding operation is used here instead of the union operation. The union operation can be used only if two functions have disjoint domains. The disjointness of the domains of $up_{\neg\epsilon}$, up_ϵ, and up_r will be proven in Theorem 5.2.2 below.

$$\begin{aligned}
& up(l_1, l_2) \\
= & \{\text{Definition of } \oplus. \ endwithR(l_1, l_2) \text{ does not hold. }\} \\
& up_{\neg\epsilon}(l_1, l_2) \\
= & \{\text{Definition of } up_{\neg\epsilon}. \ time(last(l_1)) > time(last(l_2)) \text{ holds}\} \\
& (\langle\rangle, \langle last(l_1)\rangle) \\
= & \{\text{Definition of } last.\} \\
& (\langle\rangle, \langle D(x)_1\rangle)
\end{aligned}$$

So $D(x)$ should be applied to S_2 in synchronization.

To show that the *up* function can be used for synchronizing any logs, it must be proven that *up* can be applied to any given pair of logs. Moreover, the compositional approach of defining *up* makes sense only if the domains of its component functions are disjoint. These two properties are formalized in the following two theorems.

Theorem 5.2.1 (Totality). *up is a total function.*

Proof. It is sufficient to show that

$$\operatorname{dom} up = OneKeyLog \times OneKeyLog,$$

in order to prove that *up* is a total function.

$$\begin{aligned}
& \operatorname{dom} up \\
= & \{\text{Definition of } \oplus. \ \} \\
& \operatorname{dom} up_{\neg\epsilon} \cup \operatorname{dom} up_{\epsilon} \cup \operatorname{dom} up_r \\
= & \{\text{Definitions of } up_{\neg\epsilon} \text{ and } up_{\epsilon}.\} \\
& \{(l_1, l_2) : OneKeyLog \times OneKeyLog \mid \neg endwithR(l_1, l_2) \wedge \\
& \qquad last_{\epsilon}(l_1) \neq \epsilon \wedge last_{\epsilon}(l_2) \neq \epsilon\} \cup \\
& \{(l_1, l_2) : OneKeyLog \times OneKeyLog \mid \neg endwithR(l_1, l_2) \wedge \\
& \qquad (last_{\epsilon}(l_1) = \epsilon \vee last_{\epsilon}(l_2) = \epsilon)\} \cup \operatorname{dom} up_r \\
= & \{\text{Set theory. Predicate logic.}\} \\
& \{(l_1, l_2) : OneKeyLog \times OneKeyLog \mid \neg endwithR(l_1, l_2)\} \cup \operatorname{dom} up_r \\
= & \{\text{Definitions of } up_r.\} \\
& \{(l_1, l_2) : OneKeyLog \times OneKeyLog \mid \neg endwithR(l_1, l_2)\} \cup \\
& \{(l_1, l_2) : OneKeyLog \times OneKeyLog \mid endwithR(l_1, l_2)\} \\
= & \{\text{Set theory. Predicate logic.}\} \\
& \{(l_1, l_2) : OneKeyLog \times OneKeyLog \mid true\} \\
= & OneKeyLog \times OneKeyLog
\end{aligned}$$

□

Theorem 5.2.2 (Disjointness). *The domains of $up_{\neg\epsilon}$, up_ϵ and up_r are disjoint.*

Proof. In order to prove the disjointness, it is sufficient to prove

(1) $\mathrm{dom}\, up_\epsilon \subseteq \mathrm{dom}\, up \setminus \mathrm{dom}\, up_r$

(2) $\mathrm{dom}\, up_{\neg\epsilon} \subseteq (\mathrm{dom}\, up \setminus \mathrm{dom}\, up_r) \setminus \mathrm{dom}\, up_\epsilon$

The proof of (1).

$$\begin{aligned}
&\mathrm{dom}\, up \setminus \mathrm{dom}\, up_r\\
=\ &\{\text{Definitions of } up \text{ and } up_r. \text{ Totality of } up. \text{ Definition of } \setminus.\}\\
&\{(l_1, l_2) : \mathit{OneKeyLog} \times \mathit{OneKeyLog} \mid \neg \mathit{endwithR}(l_1, l_2)\}\\
\supseteq\ &\{\text{Predicate logic.}\}\\
&\{(l_1, l_2) : \mathit{OneKeyLog} \times \mathit{OneKeyLog} \mid \neg \mathit{endwithR}(l_1, l_2) \wedge\\
&\qquad (\mathit{last}_\epsilon(l_1) = \epsilon \vee \mathit{last}_\epsilon(l_2) = \epsilon)\}\\
=\ &\{\text{Definition of } up_\epsilon.\}\\
&\mathrm{dom}\, up_\epsilon
\end{aligned}$$

The proof of (2).

$$\begin{aligned}
&(\mathrm{dom}\, up \setminus \mathrm{dom}\, up_r) \setminus \mathrm{dom}\, up_\epsilon\\
=\ &\{\text{Definitions of } up \text{ and } up_r. \text{ Totality of } up. \text{ Definition of } \setminus.\}\\
&\{(l_1, l_2) : \mathit{OneKeyLog} \times \mathit{OneKeyLog} \mid \neg \mathit{endwithR}(l_1, l_2)\} \setminus \mathrm{dom}\, up_\epsilon\\
=\ &\{\text{Definition of } up_\epsilon. \text{ Definition of } \setminus.\}\\
&\{(l_1, l_2) : \mathit{OneKeyLog} \times \mathit{OneKeyLog} \mid \neg \mathit{endwithR}(l_1, l_2) \wedge\\
&\qquad \mathit{last}_\epsilon(l_1) \neq \epsilon \wedge \mathit{last}_\epsilon(l_2) \neq \epsilon\}\\
=\ &\{\text{Definition of } up_{\neg\epsilon}.\}\\
&\mathrm{dom}\, up_{\neg\epsilon}
\end{aligned}$$

□

5.2.2 No-update-loss

When the no-update-loss rule is used, the most recent modification is chosen for propagation in synchronization. To this effect, some basic definitions will first be introduced. Given a log, the predicate *modified* checks whether the log records an update access.

$$\begin{array}{|l}
\mathit{modified}\_ : \mathbb{P}(\mathit{OneKeyLog})\\
\hline
\forall\, l : \mathit{OneKeyLog} \bullet \mathit{modified}(l) \Leftrightarrow\\
\qquad (\exists\, i : \mathrm{dom}\, l \bullet \mathit{op}(l(i)) = \mathit{write} \vee \mathit{op}(l(i)) = \mathit{add})
\end{array}$$

In this case the value of the modified data item is relevant. Since the value of the data item before the data item is deleted is used in the rule, the recorded delete accesses are not considered to be modifications. Though an add access does not provide a valid value of the data item, such an access can be regarded as a modification.

Given a log, the partial function *lastwrite* retrieves the most recent modification from the log, if any.

$$\begin{array}{|l}
\mathit{lastwrite} : \mathit{OneKeyLog} \nrightarrow \mathit{LogEntry} \\
\hline
\operatorname{dom} \mathit{lastwrite} = \{ l : \mathit{OneKeyLog} \mid \mathit{modified}(l) \} \\
\mathit{lastwrite} = \lambda\, l : \mathit{OneKeyLog} \bullet \\
\quad \textbf{if } \mathit{op}(\mathit{last}(l)) = \mathit{write} \lor \mathit{op}(\mathit{last}(l)) = \mathit{add} \textbf{ then } \mathit{last}(l) \\
\quad \textbf{else } \mathit{lastwrite}(\mathit{front}(l))
\end{array}$$

The no-update-loss rule can be defined by the following partial functions. nup_{ww} specifies that if both logs are modified, the last modification is chosen for propagation. nup_w describes that if only one log is modified, this modification is chosen. $nup_{\neg w}$ describes that if none of the logs is modified, the more recent operation is chosen in synchronization. nup_ϵ and nup_r deal with the remaining cases, such as empty logs and logs ending with read accesses.

$$\begin{array}{|l}
\mathit{nup}_{ww} : \mathit{RULE} \\
\hline
\operatorname{dom} \mathit{nup}_{ww} = \{ (l_1, l_2) : \mathit{OneKeyLog} \times \mathit{OneKeyLog} \mid \neg \mathit{endwithR}(l_1, l_2) \land \\
\quad \mathit{modified}(l_1) \land \mathit{modified}(l_2) \} \\
\mathit{nup}_{ww} = \lambda\, l_1, l_2 : \mathit{OneKeyLog} \bullet \\
\quad \textbf{if } \mathit{time}(\mathit{lastwrite}(l_1)) < \mathit{time}(\mathit{lastwrite}(l_2)) \\
\quad \textbf{then } (\langle \mathit{lastwrite}(l_2) \rangle, \langle\rangle) \\
\quad \textbf{else if } \mathit{time}(\mathit{lastwrite}(l_2)) < \mathit{time}(\mathit{lastwrite}(l_1)) \\
\qquad \textbf{then } (\langle\rangle, \langle \mathit{lastwrite}(l_1) \rangle) \\
\qquad \textbf{else } (\langle\rangle, \langle\rangle)
\end{array}$$

$$\begin{array}{|l}
\mathit{nup}_{w} : \mathit{RULE} \\
\hline
\operatorname{dom} \mathit{nup}_{w} = \{ (l_1, l_2) : \mathit{OneKeyLog} \times \mathit{OneKeyLog} \mid \neg \mathit{endwithR}(l_1, l_2) \land \\
\quad (\neg \mathit{modified}(l_1) \land \mathit{modified}(l_2)) \lor (\mathit{modified}(l_1) \land \neg \mathit{modified}(l_2)) \} \\
\mathit{nup}_{w} = \lambda\, l_1, l_2 : \mathit{OneKeyLog} \bullet \\
\quad \textbf{if } \mathit{modified}(l_1) \textbf{ then } (\langle\rangle, \langle \mathit{lastwrite}(l_1) \rangle) \textbf{ else } (\langle \mathit{lastwrite}(l_2) \rangle, \langle\rangle)
\end{array}$$

$$\begin{array}{|l}
\mathit{nup}_{\neg w} : \mathit{RULE} \\
\hline
\operatorname{dom} \mathit{nup}_{\neg w} = \{ (l_1, l_2) : \mathit{OneKeyLog} \times \mathit{OneKeyLog} \mid \neg \mathit{endwithR}(l_1, l_2) \land \\
\quad \neg \mathit{modified}(l_1) \land \neg \mathit{modified}(l_2) \land \mathit{last}_\epsilon(l_1) \neq \epsilon \land \mathit{last}_\epsilon(l_2) \neq \epsilon \} \\
\mathit{nup}_{\neg w} = \lambda\, l_1, l_2 : \mathit{OneKeyLog} \bullet \\
\quad \textbf{if } \mathit{time}(\mathit{last}(l_1)) < \mathit{time}(\mathit{last}(l_2)) \textbf{ then } (\langle \mathit{last}(l_2) \rangle, \langle\rangle) \\
\quad \textbf{else if } \mathit{time}(\mathit{last}(l_2)) < \mathit{time}(\mathit{last}(l_1)) \textbf{ then } (\langle\rangle, \langle \mathit{last}(l_1) \rangle) \\
\qquad \textbf{else } (\langle\rangle, \langle\rangle)
\end{array}$$

$nup_\epsilon : RULE$

```
dom $nup_\epsilon = \{(l_1, l_2) : OneKeyLog \times OneKeyLog \mid \neg endwithR(l_1, l_2) \wedge$
    $\neg modified(l_1) \wedge \neg modified(l_2) \wedge (last_\epsilon(l_1) = \epsilon \vee last_\epsilon(l_2) = \epsilon)\}$
$nup_\epsilon = \lambda\, l_1, l_2 : OneKeyLog \bullet$
    **if** $last_\epsilon(l_1) \neq \epsilon$ **then** $(\langle\rangle, \langle last(l_1)\rangle)$
    **else if** $last_\epsilon(l_2) \neq \epsilon$ **then** $(\langle last(l_2)\rangle, \langle\rangle)$
        **else** $(\langle\rangle, \langle\rangle)$
```

$nup_r : RULE$

```
dom $nup_r = \{(l_1, l_2) : OneKeyLog \times OneKeyLog \mid endwithR(l_1, l_2)\}$
$nup_r = \lambda\, l_1, l_2 : OneKeyLog \bullet$
    **if** $last_\epsilon(l_1) = read \wedge last_\epsilon(l_2) = read$ **then** $nup(front(l_1), front(l_2))$
    **else if** $last_\epsilon(l_1) = read$ **then** $nup(front(l_1), l_2)$
        **else** $nup(l_1, front(l_2))$
```

Thus, the full specification of the no-update-loss rule can be defined as follows.

$$nup \mathrel{\widehat{=}} nup_{ww} \oplus nup_w \oplus nup_{\neg w} \oplus nup_\epsilon \oplus nup_r.$$

For example, let l_1 and l_2 be the logs of the data spaces S_1 and S_2 in Figure 5.1, respectively.

$$l_1 = \langle A(x)_1,\, W(x)1_1, R(x)1_1,\, W(x)2_1, D(x)_1\rangle$$
$$l_2 = \langle A(x)_2,\, W(x)1_2, R(x)1_2,\, W(x)3_2\rangle$$

In this case, $time(W(x)2_1) < time(W(x)3_2)$ holds. Synchronization can be effected by applying the nup function to l_1 and l_2 as follows.

$nup(l_1, l_2)$

$=$ {Definition of $\oplus$. $endwithR(l_1, l_2)$ does not hold. $modified(l_1)$ and $modified(l_2)$ hold. }

$nup_{ww}(l_1, l_2)$

$=$ {Definition of nup_{ww}. $time(lastwrite(l_1)) < time(lastwrite(l_2))$ holds, where $lastwrite(l_1) = W(x)2_1$ and $lastwrite(l_2) = W(x)3_2$.}

$(\langle lastwrite(l_2)\rangle, \langle\rangle)$

$=$ {Definition of $lastwrite$.}

$(\langle W(x)3_2\rangle, \langle\rangle)$

Thus $W(x)3$ should be applied to S_1 in synchronization while S_2 need not be changed. Applying $W(x)3_2$ to S_1 cannot be done simply by appending $W(x)3_2$ to the end of the log of S_1, since the timestamp of $W(x)3_2$ is "before" that of $D(x)_1$. Appending would otherwise violate the time ordering property of log entries. Applying $W(x)3$ to S_1 requires an additional add operation, because that x should first exist in the data space S_1 before being read, updated or deleted. Without an add access, the log of S_1 would otherwise look like

$$\langle A(x)_1, W(x)1_1, R(x)1_1, W(x)2_1, D(x)_1, W(x)3_2\rangle,$$

which is not a valid log.

In the second example, let l_1 and l_2 be the logs of the data spaces S_1 and S_2 in Figure 5.2, respectively.

$$l_1 = \langle A(x)_1, W(x)1_1, R(x)1_1, W(x)2_1, D(x)_1\rangle$$
$$l_2 = \langle A(x)_2, W(x)1_2, R(x)1_2, W(x)3_2\rangle$$

In this case, $time(W(x)2_1) > time(W(x)3_2)$ holds. Applying the function *nup* to l_1 and l_2 works as follows.

$$\begin{aligned}
& nup(l_1, l_2) \\
= \ & \{\text{Definition of } \oplus.\ endwithR(l_1, l_2) \text{ does not hold. } modified(l_1) \text{ and } \\
& \quad modified(l_2) \text{ hold. }\} \\
& nup_{ww}(l_1, l_2) \\
= \ & \{\text{Definition of } nup_{ww}.\ time(lastwrite(l_1)) > time(lastwrite(l_2)) \text{ holds,} \\
& \quad \text{where } lastwrite(l_1) = W(x)2_1 \text{ and } lastwrite(l_2) = W(x)3_2.\} \\
& (\langle\rangle, \langle lastwrite(l_1)\rangle) \\
= \ & \{\text{Definition of } lastwrite.\} \\
& (\langle\rangle, \langle W(x)2_1\rangle)
\end{aligned}$$

Thus $W(x)2$ should be applied to S_2 in synchronization while S_1 remains untouched. Note that applying $W(x)2$ to S_2 is not yet enough to achieve the consistency of x in S_1 and S_2. The deletion of x at S_1 should be "undone". For example, an add operation $add(x)$ can be applied at S_1, followed by a write operation $W(x)2$.

Like the *up* function, the totality and disjointness of *nup* need to be verified.

Theorem 5.2.3 (Totality). *nup is a total function.*

Proof. To prove the totality, it is sufficient to prove that

$$\mathrm{dom}\, nup = OneKeyLog \times OneKeyLog,$$

The proof can be obtained in a similar way as that of Theorem 5.2.1. □

Theorem 5.2.4 (Disjointness). *The domains of nup_{ww}, nup_w, $nup_{\neg w}$, nup_ϵ and nup_r are disjoint.*

Proof. To prove the disjointness, it should be proven that given a pair of logs, $l : OneKeyLog \times OneKeyLog$, l belongs to one and only one of the domains of nup_{ww}, nup_w, $nup_{\neg w}$, nup_ϵ and nup_r. The proof can be obtained in a similar way as that of Theorem 5.2.2. □

5.2.3 Weak-no-update-loss

The weak no-update-loss rule differs from the no-update-loss rule in handling deletion. The most recent modification is propagated only if it has not yet been deleted. To specify the weak no-update-loss rule, the predicate *justmodified* is first introduced. Given a log, *justmodified* checks whether there is an undeleted modification.

$$
\begin{array}{|l}
\mathit{justmodified}\_ : \mathbb{P}(\mathit{OneKeyLog}) \\
\hline
\forall\, l : \mathit{OneKeyLog} \bullet \mathit{justmodified}(l) \Leftrightarrow \\
\qquad (\exists\, i : \operatorname{dom} l \bullet (\mathit{op}(l(i)) = \mathit{write} \lor \mathit{op}(l(i)) = \mathit{add}) \land \\
\qquad\qquad (\neg \exists\, j : \operatorname{dom} l \bullet i < j \land \mathit{op}(l(j)) = \mathit{delete}))
\end{array}
$$

The weak no-update-loss rule can be specified by the following partial functions. wep_{ww} specifies that if both logs were just modified, the most recent modification will be chosen for propagation. wep_{w_1} describes the selection of log entries for propagation when the first log of the log pair records an undeleted modification and the second log of the pair records any modification. wep_{w_3} works in a similar way as wep_{w_1}. It is applied when the second log of the log pair has just been modified and the first log of the log pair was once modified. wep_{w_2} and wep_{w_4} are applied when there is a log which has not been modified at all.[4] $wep_{\neg\epsilon}$ describes that if none of the logs has been modified, the more recent operation will be chosen in synchronization. wep_{ϵ} and wep_{r} deal with the remaining cases, such as empty logs and logs ending with read accesses.

Note that the selection of log entries for propagation defined by wep_{w_1} works as follows. If the last modification in the first log happened before the last modification in the second log, the last operation of the second log will be chosen to be applied to the data space of the first log; otherwise, the last modification in the first log will be applied to the data space of the second log.[5]

[4] Note that in the definition of wep_{w_2} there is redundancy. Since $\mathit{justmodified}(l_2)$ implies $\mathit{modified}(l_2)$, $\neg \mathit{modified}(l_2)$ implies $\neg \mathit{justmodified}(l_2)$. So, it is redundant to include $\neg \mathit{justmodified}(l_2)$ in the domain definition of wep_{w_2}. The reason for this redundancy is manifested in the proof of the totality of the *wep* in Theorem 5.2.5. Likewise, a similar redundancy is introduced in the domain definition of wep_{w_4}

[5] This treatment is based on the assumption that when the deletion occurred on the data space which had been most recently updated, the deletion was taken by the user with the awareness of the most recent update. For example, a user modifies a copy of a file on a portable device while he is away from home. He later deletes the copy on the portable device. When the user synchronizes his data with a file server at home, all operations performed on the portable devices should be applied to the home server if the copy of the file on the home server was not updated. This condition holds only if the most recent update occurred on the portable device, not the home server. If the copy on the home server was modified, it is likely that this copy was modified by another home user. Thus, this modification can be propagated to the portable device. Otherwise, it can be reported to the user.

```
$wep_{ww} : RULE$
--------
$\mathrm{dom}\, wep_{ww} = \{(l_1, l_2) : OneKeyLog \times OneKeyLog \mid \neg endwithR(l_1, l_2) \wedge$
    $justmodified(l_1) \wedge justmodified(l_2)\}$
$wep_{ww} = \lambda\, l_1, l_2 : OneKeyLog \bullet$
    **if** $time(lastwrite(l_1)) < time(lastwrite(l_2))$
    **then** $(\langle lastwrite(l_2)\rangle, \langle\rangle)$
    **else if** $time(lastwrite(l_2)) < time(lastwrite(l_1))$
        **then** $(\langle\rangle, \langle lastwrite(l_1)\rangle)$
        **else** $(\langle\rangle, \langle\rangle)$
```

```
$wep_{w_1} : RULE$
--------
$\mathrm{dom}\, wep_{w_1} = \{(l_1, l_2) : OneKeyLog \times OneKeyLog \mid \neg endwithR(l_1, l_2) \wedge$
    $justmodified(l_1) \wedge \neg justmodified(l_2) \wedge modified(l_2)\}$
$wep_{w_1} = \lambda\, l_1, l_2 : OneKeyLog \bullet$
    **if** $time(lastwrite(l_1)) < time(lastwrite(l_2))$
    **then** $(\langle last(l_2)\rangle, \langle\rangle)$
    **else if** $time(lastwrite(l_2)) < time(lastwrite(l_1))$
        **then** $(\langle\rangle, \langle lastwrite(l_1)\rangle)$
        **else** $(\langle\rangle, \langle\rangle)$
```

```
$wep_{w_2} : RULE$
--------
$\mathrm{dom}\, wep_{w_2} = \{(l_1, l_2) : OneKeyLog \times OneKeyLog \mid \neg endwithR(l_1, l_2) \wedge$
    $justmodified(l_1) \wedge \neg justmodified(l_2) \wedge \neg modified(l_2)\}$
$wep_{w_2} = \lambda\, l_1, l_2 : OneKeyLog \bullet (\langle\rangle, \langle lastwrite(l_1)\rangle)$
```

```
$wep_{w_3} : RULE$
--------
$\mathrm{dom}\, wep_{w_3} = \{(l_1, l_2) : OneKeyLog \times OneKeyLog \mid \neg endwithR(l_1, l_2) \wedge$
    $justmodified(l_2) \wedge \neg justmodified(l_1) \wedge modified(l_1)\}$
$wep_{w_3} = \lambda\, l_1, l_2 : OneKeyLog \bullet$
    **if** $time(lastwrite(l_1)) < time(lastwrite(l_2))$
    **then** $(\langle lastwrite(l_2)\rangle, \langle\rangle)$
    **else if** $time(lastwrite(l_2)) < time(lastwrite(l_1))$
        **then** $(\langle\rangle, \langle last(l_1)\rangle)$
        **else** $(\langle\rangle, \langle\rangle)$
```

```
$wep_{w_4} : RULE$
--------
$\mathrm{dom}\, wep_{w_4} = \{(l_1, l_2) : OneKeyLog \times OneKeyLog \mid \neg endwithR(l_1, l_2) \wedge$
    $justmodified(l_2) \wedge \neg justmodified(l_1) \wedge \neg modified(l_1)\}$
$wep_{w_4} = \lambda\, l_1, l_2 : OneKeyLog \bullet (\langle lastwrite(l_2)\rangle, \langle\rangle)$
```

```
$wep_{\neg\epsilon} : RULE$
────────────
dom $wep_{\neg\epsilon} = \{(l_1, l_2) : OneKeyLog \times OneKeyLog \mid \neg endwithR(l_1, l_2) \wedge$
    $\neg justmodified(l_1) \wedge \neg justmodified(l_2) \wedge last_\epsilon(l_1) \neq \epsilon \wedge last_\epsilon(l_2) \neq \epsilon\}$
$wep_{\neg\epsilon} = \lambda\, l_1, l_2 : OneKeyLog \bullet$
    **if** $time(lastwrite(l_1)) < time(lastwrite(l_2))$
    **then** $(\langle lastwrite(l_2)\rangle, \langle\rangle)$
    **else if** $time(lastwrite(l_2)) < time(lastwrite(l_1))$
        **then** $(\langle\rangle, \langle lastwrite(l_1)\rangle)$
        **else** $(\langle\rangle, \langle\rangle)$
```

```
$wep_\epsilon : RULE$
────────────
dom $wep_\epsilon = \{(l_1, l_2) : OneKeyLog \times OneKeyLog \mid \neg endwithR(l_1, l_2) \wedge$
    $\neg justmodified(l_1) \wedge \neg justmodified(l_2) \wedge last_\epsilon(l_1) = \epsilon \vee last_\epsilon(l_2) = \epsilon\}$
$wep_\epsilon = \lambda\, l_1, l_2 : OneKeyLog \bullet$
    **if** $last_\epsilon(l_1) \neq \epsilon$ **then** $(\langle\rangle, \langle last(l_1)\rangle)$
    **else if** $last_\epsilon(l_2) \neq \epsilon$ **then** $(\langle last(l_2)\rangle, \langle\rangle)$
        **else** $\langle\rangle$
```

```
$wep_r : RULE$
────────────
dom $wep_r = \{(l_1, l_2) : OneKeyLog \times OneKeyLog \mid endwithR(l_1, l_2)\}$
$wep_r = \lambda\, l_1, l_2 : OneKeyLog \bullet$
    **if** $last_\epsilon(l_1) = read \wedge last_\epsilon(l_2) = read$ **then** $wep(front(l_1), front(l_2))$
    **else if** $last_\epsilon(l_1) = read$ **then** $wep(front(l_1), l_2)$
        **else** $wep(l_1, front(l_2))$
```

The full specification of the weak no-update-loss rule can be defined as follows.

$$wep \mathrel{\widehat=} wep_{ww} \oplus wep_{w_1} \oplus wep_{w_2} \oplus wep_{w_3} \oplus wep_{w_4} \oplus wep_{\neg\epsilon} \oplus wep_\epsilon \oplus wep_r.$$

For example, let l_1 and l_2 be the logs of the data spaces S_1 and S_2 in Figure 5.1, respectively.

$$l_1 = \langle A(x)_1,\, W(x)1_1, R(x)1_1,\, W(x)2_1, D(x)_1\rangle$$
$$l_2 = \langle A(x)_2,\, W(x)1_2, R(x)1_2,\, W(x)3_2\rangle$$

Note that $time(W(x)2_1) < time(W(x)3_2)$ holds. Synchronization can be effected by applying the *wep* function to l_1 and l_2 as follows.

$wep(l_1, l_2)$

$=$ {Definition of $\oplus$. $justmodified(l_2)$, $\neg justmodified(l_1)$ and $modified(l_1)$ hold. }

$wep_{w_3}(l_1, l_2)$

$= \{$Definition of wep_{w_3}. $time(lastwrite(l_1)) < time(lastwrite(l_2))$ holds, where $lastwrite(l_1) = W(x)2_1$ and $lastwrite(l_2) = W(x)3_2$.$\}$
$(\langle lastwrite(l_2)\rangle, \langle\rangle)$
$= \{$Definition of $lastwrite$.$\}$
$(\langle W(x)3_2\rangle, \langle\rangle)$

Thus $W(x)3$ should be applied to S_1 in synchronization while S_2 remains untouched.

Let l_1 and l_2 be the logs of the data spaces S_1 and S_2 in Figure 5.2, respectively.

$$l_1 = \langle A(x)_1, W(x)1_1, R(x)1_1, W(x)2_1, D(x)_1\rangle$$
$$l_2 = \langle A(x)_2, W(x)1_2, R(x)1_2, W(x)3_2\rangle$$

In this case, $time(W(x)2_1) > time(W(x)3_2)$. Synchronization can be effected by applying the *wep* function to l_1 and l_2 as follows.

$wep(l_1, l_2)$
$= \{$Definition of $\oplus$. $justmodified(l_2)$, $\neg justmodified(l_1)$ and $modified(l_1)$ hold. $\}$
$wep_{w_3}(l_1, l_2)$
$= \{$Definition of wep_{w_3}. $time(lastwrite(l_1)) > time(lastwrite(l_2))$ holds, where $lastwrite(l_1) = W(x)2_1$ and $lastwrite(l_2) = W(x)3_2$.$\}$
$(\langle\rangle, \langle last(l_1)\rangle)$
$= \{$Definition of $last$.$\}$
$(\langle\rangle, \langle (D(x))_1\rangle)$

So $D(x)$ should be applied to S_2 in synchronization.

As with the *up* and *nup* functions , the totality and disjointness of *wep* need to be verified.

Theorem 5.2.5 (Totality). *wep is a total function.*

Proof. To prove the totality, it is sufficient to prove that

$$\text{dom}\, wep = OneKeyLog \times OneKeyLog.$$

The proof can be obtained in a similar way as that of Theorem 5.2.1. □

Theorem 5.2.6 (Disjointness). *The domains of wep_{ww}, wep_{w_1}, wep_{w_2}, wep_{w_3}, wep_{w_4}, $wep_{\neg\epsilon}$, wep_{ϵ}, and wep_r are disjoint.*

Proof. To prove the disjointness, it should be proven that given a pair of logs, $l : OneKeyLog \times OneKeyLog$, l belongs to one and only one of the domains of wep_{ww}, wep_{w_1}, wep_{w_2}, wep_{w_3}, wep_{w_4}, $wep_{\neg\epsilon}$, wep_{ϵ}, and wep_r. The proof can be obtained in a similar way as that of Theorem 5.2.2. □

5.3 Soundness of using characteristic-entry logs

When applying the semantic rules formally specified in the previous section, characteristic-entry logs can be used in the same way as normal logs. Due to the fact that normal logs can be converted to characteristic-entry logs, it can also be proven that normal logs and characteristic-entry logs can be mixed: in data synchronization, a data space can use a normal log while another data space can use a characteristic-entry log.

5.3.1 Up-to-date

Characteristic-entry logs can be used when applying the up-to-date rule in data synchronization. To prove this, two lemmas will first be proven.

Lemma 5.3.1. *Given any log, $l : OneKeyLog$, $last_\epsilon(l) = last_\epsilon(ce(l))$.*

Proof. According to the definitions of $last_\epsilon$ and ce, this property can be easily proven. □

Lemma 5.3.2. *Given any two logs, $l_1, l_2 : OneKeyLog$, $endwithR(l_1, l_2) \Leftrightarrow endwithR(ce(l_1), ce(l_2))$.*

Proof. According to the definitions of *endwithR*, *ce*, and Lemma 5.3.1, this property can be easily proven. □

The following theorem says that characteristic-entry logs can be used when applying the up-to-date rule in data synchronization.

Theorem 5.3.1 (Soundness). *Given any two logs, $l_1, l_2 \in OneKeyLog$, $up(l_1, l_2) = up(ce(l_1), ce(l_2))$.*

Proof. *up* is defined in a compositional way. Its domain consists of the domains of $up_{\neg\epsilon}$, up_ϵ and up_r, which are disjoint from each other. So the proof of this theorem can be obtained case by case.

Case $up_{\neg\epsilon}$. In this case, none of l_1 and l_2 is empty. According Lemmas 5.3.1 and 5.3.2, it can be easily verified that

$$(ce(l_1), ce(l_2)) \in \mathrm{dom}\, up_{\neg\epsilon}.$$

According to the definition of $up_{\neg\epsilon}$, the most recent one of $last(l_1)$ and $last(l_2)$ is chosen for propagation, which is the most recent one of $last(ce(l_1))$ and $last(ce(l_2))$. Thus $up_\epsilon(l_1, l_2) = up_\epsilon(ce(l_1), (l_2))$. Case up_ϵ. In a similar way, the case of up_ϵ can be proven. Case up_r. up_r filters out read accesses in logs and applies *up* to the residual log entries. When *up* is applied to the residual log entries, $up_{\neg\epsilon}$ and up_ϵ are used. Due to the fact that the cases of up_ϵ and $up_{\neg\epsilon}$ have been proven, it can be easily proven that $up_r(l_1, l_2) = up_r(ce(l_1), ce(l_2))$. □

Theorem 5.3.2. *Given any two logs, $l_1, l_2 \in OneKeyLog$,*

$$up(l_1, l_2) = up(ce(l_1), l_2)$$
$$up(l_1, l_2) = up(l_1, ce(l_2))$$

Proof. The proof of this theorem can be obtained in a similar way as that of Theorem 5.3.1. □

5.3.2 No-update-loss

Two lemmas will first be introduced.

Lemma 5.3.3. *Given a log, $l : OneKeyLog$, $modified(l) = modified(ce(l))$.*

Proof. According to the definitions of *modified* and *ce*, this property can be easily proven. □

Lemma 5.3.4. *Given a log, $l : OneKeyLog$, $lastwrite(l) = lastwrite(ce(l))$.*

Proof. According to the definitions of *lastwrite* and *ce*, this property can be easily proven. □

Characteristic-entry logs can be used when applying the no-update-loss rule in data synchronization, which is stated in the following theorem.

Theorem 5.3.3 (Soundness). *Given any two logs, $l_1, l_2 \in OneKeyLog$, $nup(l_1, l_2) = nup(ce(l_1), ce(l_2))$.*

Proof. The proof of this theorem can be obtained in a similar way as that of Theorem 5.3.1. □

Theorem 5.3.4. *Given any two logs, $l_1, l_2 \in OneKeyLog$,*

$$nup(l_1, l_2) = nup(ce(l_1), l_2)$$
$$nup(l_1, l_2) = nup(l_1, ce(l_2))$$

Proof. The proof of this theorem can be obtained in a similar way as that of Theorem 5.3.1. □

5.3.3 Weak-no-update-loss

Lemma 5.3.5. *Given a log,* $l : OneKeyLog$*,*
$justmodified(l) = justmodified(ce(l))$*.*

Proof. According to the definitions of *modified* and *ce*, this property can be easily proven. □

Characteristic-entry logs can be used when applying the weak no-update-loss rule in data synchronization, which is stated in the following theorem.

Theorem 5.3.5 (Soundness). *Given any two logs,* $l_1, l_2 \in OneKeyLog$*,*
$wep(l_1, l_2) = wep(ce(l_1), ce(l_2))$*.*

Proof. The proof of this theorem can be obtained in a similar way as that of Theorem 5.3.1. □

Theorem 5.3.6. *Given any two logs,* $l_1, l_2 \in OneKeyLog$*,*

$$wep(l_1, l_2) = wep(ce(l_1), l_2),$$
$$wep(l_1, l_2) = wep(l_1, ce(l_2)).$$

Proof. The proof of this theorem can be obtained in a similar way as that of Theorem 5.3.1. □

5.4 Concluding remarks

So far little use has been made of formal methods in studying disconnected updates. Semantic rules have always been expressed in an informal way and data synchronization was consequently rather vulnerable. In this chapter, data synchronization has been treated formally. First, semantic rules that are often used in data synchronization were formalized. Each rule was modelled as a function that takes a pair of logs and returns the log entries that need to be propagated in data synchronization. Next, it was proven that characteristic-entry logs contain sufficient information for data synchronization with respect to the rules that were studied. This provides a formal argument that characteristic-entry logs can be used in a similar way as normal logs in data synchronization.

In this work, the Z notation was used in formalization and proof. Axiomatic definitions of the notation were deliberately extensively used while schemas and schema calculus were avoided to make the specification and proof tasks easier. To manage the complexity of specifications and proofs, a compositional approach was used to define complex functions, namely using the built-in overriding operation of Z to compose partially defined functions.

This practice turns out to be rather effective. Specifications of complex functions become comprehensible. Z proves to be a powerful tool in modelling logs and semantic rules and in proving correctness of data synchronization. The mathematical machinery of Z is sufficient for this type of research projects. In the chapter, the proofs were obtained as rigorously as possible. It would be a good idea to check them using theorem provers in the future.

Part III

System design and implementation

6. The MemorySafe system

The MemorySafe system is a distributed digital asset management system for home environments. In this chapter, user requirements of managing digital assets in home environments will be described first. Next, the mechanisms that were designed in the MemorySafe system to address those user requirements will be explained. Finally, experiences regarding the implementation of the MemorySafe system will be presented.

6.1 Requirements

The MemorySafe system is intended for managing personal digital assets, such as digital photos, which are stored in different devices in home environments. The design of the system addresses the following issues that people may encounter when handling their digital photos.

6.1.1 Managing data copies

People often make copies of digital photos for a variety of reasons. For example, people usually keep high-resolution digital photos for archiving purposes. In addition, people may make copies in the following situations.

- People make photo albums, with or without the help of software tools such as Ulead Photo Explorer [161] and FlipAlbum [40]. Sometimes people make dedicated photo albums, depending on the persons to whom they are going to show the albums. One photograph may have several copies in different albums.
- To order prints of some photos, people often have to make a scaled version of the photos before sending them to a print shop. Existing digital photo printing services have restrictions on the size of images to be printed.
- To send digital photos to friends via email, people often make a smaller version of the photos. Images with a resolution of 72 dots per inch (dpi) are sufficient for on-screen displaying.
- People maintain working copies of photos. When editing a photo, a user may crop it, change brightness and contrast, adjust color saturation, add titles and so on. The resulting image is stored in a new file, next to the original file, for later retrieval.

- To improve the availability, people make copies of digital photos and upload them to portable devices, storage media, web directories or web sites.

Digital photos and their copies are treated as separate files in file systems. A user has to keep track of all the copies of a photo and must be capable of distinguishing the different copies. Usually people do so by giving indicative names to files and directories. This task becomes cumbersome as the number of digital photos and that of the devices involved increase. Therefore, the following requirement is identified.

R1. (Handling data copies). *A user should be able to easily manage logically-related data.*

6.1.2 Managing metadata

Metadata are used to describe various aspects of digital photos, such as camera settings, image properties and image descriptions. Depending on coding standards, some metadata are embedded in photo files. For example, when an image is saved in the JPG format, metadata such as category and keywords can be stored in a JPG file, together with image data.

Some metadata are not covered by coding standards. They can hence not be embedded in image files. Especially user-dependent metadata of photos, such as annotations, are often stored in separate files. Such a separation causes problems when a user moves or copies a collection of photos from one device to another. The user must be able to migrate all files relating to the photos. Otherwise, part of the descriptive information will be lost. Therefore, the following requirement is identified.

R2. (Avoiding metadata loss). *A user should be able to migrate data without metadata loss.*

Moreover, each file has its own metadata. For a digital photo with several copies, a user has to specify the metadata of individual copies repeatedly, which may introduce inconsistencies. Therefore, the following requirement is identified.

R3. (Sharing metadata among data copies). *A user should be able to manage metadata of different copies of a photo.*

6.1.3 Structuring data

To manage an increasing number of digital photos, people organize photos in different ways. For example, photos are stored in directories, which are named after location, time or events in file systems. People also make digital albums with the help of software tools. Consequently, as the number of albums and directories storing photo files increases, people will start wondering what is the best way of structuring albums?

The problem of structuring albums is linked with the problem of managing copies. As mentioned above, copies of photos are made to compose different albums. So an increase in the number of albums leads to a corresponding increase in the number of data copies.

R4. (Structuring data). *A user should be able to manage a large number of albums and photos in a flexible and coherent way, without introducing unnecessary data copies.*

With respect to this requirement, the "broken link" problem needs to be addressed as well. When composing new photo albums for slideshows or browsing, people often create links or shortcuts to their photos. Such links may be broken when the original photo files are moved to other locations. So in providing flexible methods for users to structure their photos, broken links should be avoided.

6.1.4 Managing distributed data

Many people tend to associate information with locations. Locations have different meanings for different people. Examples of locations are:

- Geographic places, such as homes and offices.
- Devices, such as desktop computers, laptop computers and PDAs. In the future, new devices will appear, such as Sepias, digital photo frames and objects with storage of the Phenom project [33].
- Conceptual locations, such as web directories.

Such locations provide powerful metaphors for users dealing with their photos. Instead of being hidden from users, locations need to be represented in a proper manner in systems.

Technically, the concept of "location of data" should not be completely hidden from users either. Data location often impacts data availability. A device can be disconnected from the Internet anytime at a user's will. To be sure that his data will be available whenever needed, a user must know for sure that the requested data have been stored on a desired device before he disconnects the device from the others.

As the number of photos and albums and that of devices increase, answering a question like "what is stored where?" will be a difficult task for a user. People may very easily forget in which device a specific album or photo is stored. Therefore, proper conceptual models of data location are required to alleviate the burden of handling distributed large data collections. The following requirement is identified.

R5. (Managing distributed data). *A user should be able to locate and manage data stored in different devices.*

Table 6.1. The user requirements that need to be addressed in the design of the MemorySafe system.

Requirement No.	Requirement description
R1	Handling data copies
R2	Avoiding metadata loss
R3	Sharing medadata among data copies
R4	Structuring data
R5	Managing distributed data
R6	Synchronizing data
R7	Managing visiting devices

6.1.5 Disconnected updates

While locating different data copies is a difficult task, keeping different data copies consistent is also a challenge. Many consumer electronic devices can function without being connected to the Internet or other devices. A user can continue to work on his or her documents, enjoy music or browse photos on the move when using portable devices.

A user may access and modify a copy of data in a device, independent of the copies in the other devices. For example, a user organizes photo albums or he modifies a working document. Disconnected updates introduce data inconsistencies. As for digital photos, both the photos and their metadata may suffer from this consistency problem.

R6. (Data synchronization). *A user should be able to manage the consistency of data copies that are stored in different locations.*

6.1.6 Accessing visiting devices

Home environments call for open systems. New devices are added and old ones are removed. Moreover, visitors may bring their own devices to share information. Therefore, an HDMS should provide support to allow such ad-hoc communications.

R7. (Managing visiting devices). *A user should be able to manage and access visiting devices in his or her home environment.*

6.1.7 Summary

Table 6.1 summarizes the identified user requirements of the MemorySafe system. When designing a real system for practical use, there are other requirements such as real-time performance, scalability and security. Since the MemorySafe system is only a prototype system, its design focuses on concept demonstration.

6.2 Design of the MemorySafe system

In the MemorySafe system, the following mechanisms are used to address the above-mentioned requirements. In the first place, *a logical identity* is assigned to each digital photograph. The logical identity of a digital photo is permanent and immutable. The logical identity is shipped with image data of the photo during data migration.

Secondly, *data aggregation* is used to address the requirements relating to multiple copies, metadata loss in data migration, and metadata sharing of data copies. Different copies and metadata of a photo are aggregated to a *resource*, a logical data unit that encapsulates all aspects of the digital photo. Data copies of a digital photo are logically stored in the same resource and can be accessed through the same entry, which avoids the separation of data copies and facilitates the management of the copied data.

Combining metadata with image data of a digital photo helps to achieve metadata sharing among different copies. When a resource is used as the basic unit in data migration, metadata loss is avoided. Furthermore, with the help of unique identifications of resources, resources that contain the same digital photo but are stored in different devices can be reliably recognized in open systems.

Thirdly, a new *linking* mechanism is used to structure digital photos and albums. A user views the MemorySafe system as a rooted acyclic directed graph. One digital photo or collection of photos can appear in several places in the data hierarchy of MemorySafe, without new copies of the photo or the photo collection being introduced. This addresses the requirement of flexible structuring.

Fourthly, *server grouping* and *server-translucency* are used in the MemorySafe system to address data-distribution-related requirements. Server grouping helps a user to define the scope of his personal information system and to access and manage visiting devices. Server-translucency, a mixture of *server-aware* and *server-transparent* data access, provides support for developing applications with different requirements relating to location awareness. Server-aware data access allows users and programmers to access data per device and can be used in data monitoring and data migration. Server-transparent data access provides a naming mechanism with which a user or a program can access data regardless where their data are stored.

Finally, *identity-based history synchronization*, which was discussed in Section 2.5, helps to manage the consistency of distributed resources in the MemorySafe system. Each resource has an access history, which records all disconnected updates of this resource. A resource can be made available on different devices. With unique identifications, copies of the resource stored on different devices can be traced back. In the case of inconsistencies, access histories of resource copies are used to resolve conflicting disconnected updates.

Table 6.2. Mechanisms in the MemorySafe system that address the user requirements described in Table 6.1.

Mechanisms	Requirements
Data aggregation	R1, R2, R3
Linking	R4
Data identities	R5, R6
Server grouping	R5, R7
Server translucency	R5
Identity-based history synchronization	R6

To summarize, Table 6.2 illustrates how the designed mechanisms of the MemorySafe system address the identified user requirements. In the rest of this section those mechanisms will be explained in detail in a top-down manner. First, the system architecture and the server grouping mechanism of the MemorySafe system will be explained. Next, data objects in the system will be introduced. After that, data structuring and data access will be presented. Finally, data synchronization is addressed.

6.2.1 Server grouping

The MemorySafe system is implemented as a functional component, responsible for distributed data management, in the Empire middleware platform [30]. MemorySafe consists of a number of up-running software programs, *MemorySafe servers*, which are interconnected via Empire's messaging system. MemorySafe servers serve data storage and retrieval requests and they represent storage devices in home environments, for example desktop computers, tablet computers, or future photo frames. Usually one device has at most one MemorySafe server up-running.

MemorySafe servers can be started and stopped whenever users want, which corresponds to devices on/off and entering/leaving activities. When a MemorySafe server is disconnected from the others, it continues to provide the functionality of data storage and retrieval.

Server grouping. In the MemorySafe system, MemorySafe servers are organized into administrative groups. An administrative group represents the collection of devices a user might access for personal use. MemorySafe servers may run on devices that are shared by multiple users in home environments. Therefore, multiple server groups coexist in home environments. Server groups of different users are logically disjoint in the MemorySafe system. However, a user can access any server at his or her own will.

In a server group, there are two types of servers: *super servers* and *thin servers*. Super servers are often installed on stationary devices and act as repositories storing massive data. Thin servers are often installed on portable devices and store limited amounts of data. For example, in home environments, desktop computers are super servers that store massive collections of

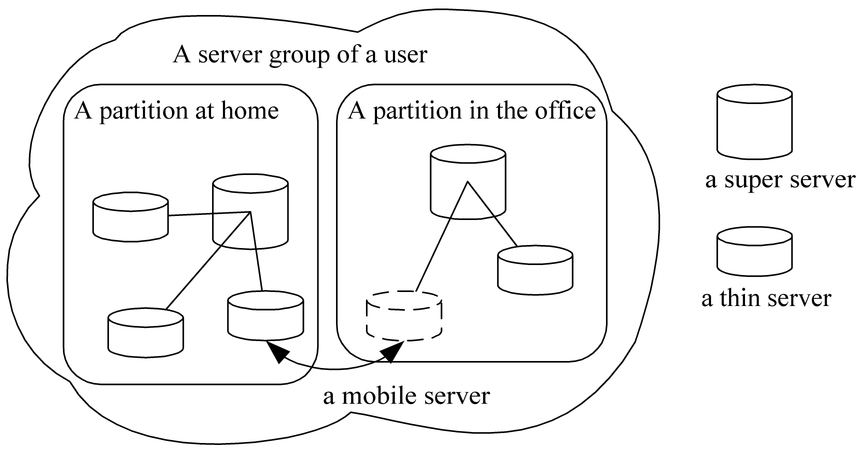


Figure 6.1. An overview of a server group.

digital photos, music and movies while tablet computers, palmtop and laptop computers are thin servers that often store parts of user data.

A server group can be extended to other environments disconnected from home environments. In this case, a server group is partitioned and thin servers can be moved from one partition to another. This is often the case for people who carry PDAs or laptops back and forth between their home and the office.

Figure 6.1 illustrates a server group belonging to a user. The server group has two partitions, each with a super server. There is a server intermittently appearing in either partition, representing a portable device that is carried by the user between home and office. Note that the following two scenarios may occur.

- There may be no super server in one partition. For example, the portable device is used on the move: super servers at home or in the office are not accessible and only the local thin server is available.
- Another possibility is that there are two or more super servers up-running in the same partition. In this situation, the super servers, as a whole, function as a fully replicated system.

In comparison with a single-server model, MemorySafe keeps physical devices visible in the system and provides flexibility in organizing devices. To minimize the complexity of managing data and programming software in multi-server systems, MemorySafe provides *server translucency* for data access. This will be discussed in Section 6.2.4. Users and programmers of the MemorySafe system can selectively work with the system in a server-aware (location-aware) or a server-transparent (location-transparent) way.

6.2.2 Logical identity and data aggregation

As mentioned in Sections 6.1.1 and 6.1.2, data copies and metadata of multimedia data objects are often stored in files different from the files storing the multimedia data objects. File systems do not usually maintain the deriving relation between original multimedia files and data copies. Nor do they preserve the relation between original multimedia files and metadata.

Resources. To solve this problem, conceptually related data of a multimedia object are aggregated into a single unit called *resource* in the MemorySafe system.[1] A resource consists of *resource identity*, *resource description*, *resource variants* and *access history*. Resources are the logical data units that a user works with in the MemorySafe system, instead of files in file systems. A resource is the basic data unit in data migration.

Resource identity. A resource obtains an identity on its creation. The identity of a resource is immutable. The identity does not change over time. When the resource is shipped to different systems, its identity will not change either. A resource identity is persistent. It survives system shutdown and crashes. The same resource identity must never be assigned to two different data objects.

In the MemorySafe system, resource identities are captured by *globally unique resource identifiers*, in short GURIs. In MemorySafe, GURIs meet the immutableness, persistence and uniqueness that are required by resource identities.[2]

One digital photo may have several copies in different devices. This is modelled in the MemorySafe system by replicating the resources modelling the photo to different MemorySafe servers. The original resource and the introduced resources share the same identity. Therefore, resources of the same digital photo can be reliably located.

Resource identities also help to avoid identity loss and resolve name clashes in data synchronization. This was one of the conclusions of Chapter 2.

Resource variants. Different copies of a digital photo are stored as data variants of a resource representing the digital photos. The data variants are treated as integral parts of the resource. For example, the original data of a digital photo and the data of other modified versions of the photo are aggregated together. A user consequently need not bother recalling the filename of a copy of the photo.

Resource variants can also be used to store application-specific data. For example, thumbnail images are often created on-the-fly for fast digital photo

[1] In the Internet architecture [102, 139, 14, 15, 16], resources can be any data objects such as static documents, dynamic links. In the MemorySafe system, resources are data objects that have a well-defined structure and are static.

[2] Since the MemorySafe system is an experimental system, it suffices to maintain the uniqueness of such an identifier within the system.

Table 6.3. The system-defined descriptors of a resource in the MemorySafe system.

Key of descriptor	Purpose
`user`	Describes the owner of a resource.
`name`	Describes the user-friendly name of a resource.
`type`	Describes the type of resource, a file resource (`dir`) or a directory resource (`nodir`).
`datatype`	Describes the data type of the data of a resource. The value of this key for example can be `txt`, `jpg`, or `bmp`.
`title`	Describes the title of an image resource.
`latest_version`	Describes the current version number of the data of a resource.
`lastmodification`	Describes the last time a resource was modified.
`latest_version_thumbnail`	Describes the current version number of the thumbnail data of a resource. The thumbnail data is regarded as a data variant and is identified by the `thumbnail`.

browsing in file systems and they are stored as temporarily created files separated from the files storing the images. The thumbnail image of a digital photo is regarded as a variant of a resource representing the photo so that the data of the thumbnail image can be stored together with that of the photo, persistently in the system. There is hence no need to generate thumbnail images at runtime. This new way of clustering data provides logical and simple views of personal data in user-centric information systems.

Resource description. A resource description is an integral part of a resource. It consists of zero or more *resource descriptors*. A resource descriptor is a key-value pair. Both parts are plain textual strings that can be read and remembered by the user. The value of the key part is unique in a resource description. In the MemorySafe system, each resource has a set of pre-defined resource descriptors, which are shown in Table 6.3. Users of the MemorySafe system can add more descriptors to a resource description.

A resource description does not only capture system-specific information; it can also be used to store semantic knowledge relating to the resource. For example, all metadata of a digital photo can be described in the resource description of a resource representing the photograph. A user may include content-related information on the photo in the resource description. Moreover, the resource description can also be used for recording use information about this resource. Such information may be how often the photo is used, what the current version is, what changes have been made since the last access, what the last operation was. Resource descriptions can be used for associative data retrieval [6, 75]

The idea of introducing extensive descriptions of files or directories is similar to that underlying the semantic file systems [49]. The idea of providing a general architecture for describing data objects is related to that underlying the Universal Resource Characteristics [139] for recording meta-information on resources in the Internet information architecture.

Access history. The concept of access history is introduced to record modifications on a resource. This treatment meets the requirement of using logs in data synchronization, which was also one of the conclusions of Chapter 2. Details of access histories will be presented in Chapter 7, along with data synchronization.

6.2.3 Linking

To allow a user to build albums of photos, *directory resources* are introduced. Directory resources contain references to other resources. To distinguish them from directory resources, resources that model digital photos will be called *file resources* here. In short, *files* and *directories* will be used for file resources and directory resources, respectively, in the rest of this chapter. They might be confused with the same terms used in file systems. Whenever there is a risk of confusion, it will be pointed out in which context the terms are used.

By using directories, a user of the MemorySafe system observes that his or her data are stored in a tree-like hierarchical structure, similar to what a user might observe in a file system of a Windows operating system.

Linking. A tree-like structure is not powerful enough to express more complex data views. For example, a user wants to make a photo album with photos from other albums. In this case, a photo will appear in multiple albums. The MemorySafe system allows a user to build up such conceptual views on his or her data, as shown in Diagram A of Figure 6.2.

The MemorySafe system allows flexible hierarchical conceptual views via *linking*: A user can create multiple *links* to a resource in different directory resources. In Diagram A of Figure 6.2, photos from the "Keukenhof' 02" and "Garden" albums are selectively associated with an album called "tulip", representing the association between those photos and a souvenir object, "tulip", in the Phenom demonstration system [33].

The MemorySafe system avoids the introduction of cycles in the system. Consider the example illustrated in Diagram B of Figure 6.2. One album contains a link to itself and another contains a reference to its parent resource. What do these mean to the user? It is just not clear. Due to the complexity of cyclic structures, it is prohibited to create a link between a resource and its parent resources in the MemorySafe system. The avoidance of cyclic structure reduces the complexity of conceptual views in the MemorySafe system.

Comparison of formal models of different linking mechanisms. Due to the introduction of links in the MemorySafe system, the conceptual view of a MemorySafe system is no longer a tree-like structure. Graph theory is

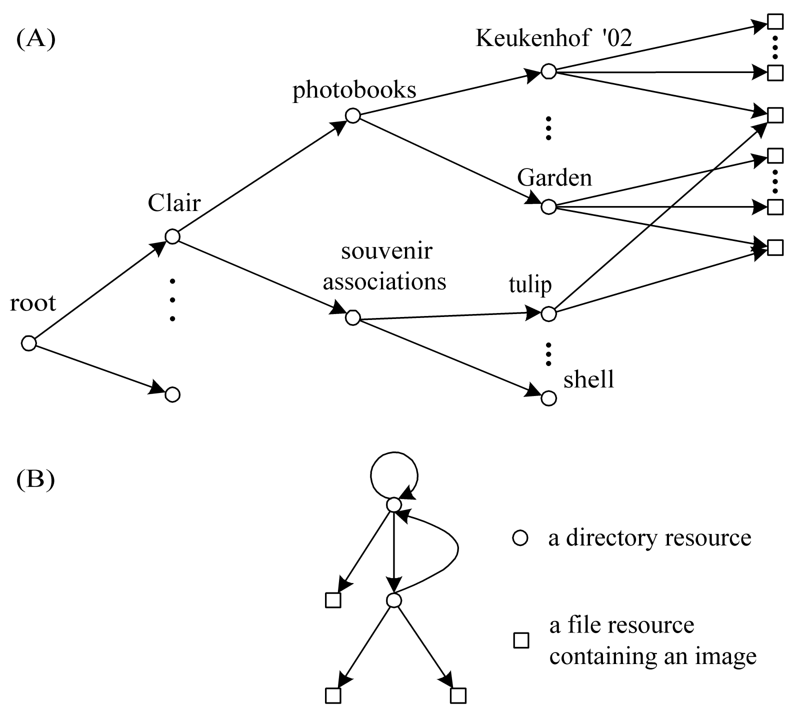


Figure 6.2. Diagram A is a conceptual view of the contents stored on a MemorySafe server. The MemorySafe system allows a resource to be referenced by multiple resources. Diagram B is a cyclic structure that could be created by links, which is however avoided in the MemorySafe system.

used here to give a more formal definition of the conceptual view of the MemorySafe system.[3]

In graph theory, a *digraph* is a set of vertices and a set of arrows that connect two vertices. Arrows are *multi-arrows* if they connect the same two vertices. A *path* is a sequence of arrows that link up with any two vertices. A digraph is *connected* if, given any two vertices, there is a path connecting them. A digraph is *acyclic* if it does not contain a path, the first and last vertices of which are the same. A digraph is *rooted* if it contains a vertex from which all paths lead away.

Conceptually, the files in the MemorySafe system form a digraph that is connected, acyclic and rooted. Unlike with the MemorySafe system, the files of a Unix file system form a digraph that is connected, acyclic, rooted and with multi-arrows. A Windows file system is a digraph that is connected, acyclic, rooted and with the property that there is a root vertex and all

[3] The terms used in this section appear in [130].

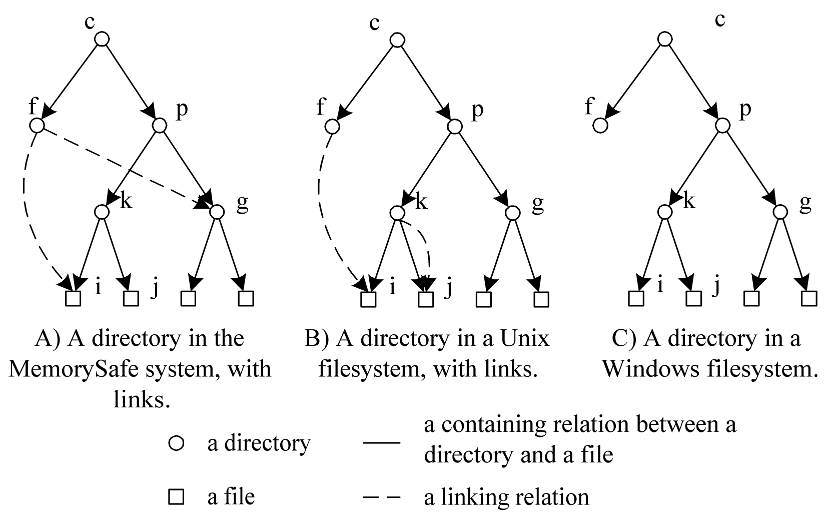

c
f
p
k
g
i
j
A) A directory in the MemorySafe system, with links.
B) A directory in a Unix filesystem, with links.
C) A directory in a Windows filesystem.
○ a directory
□ a file
a containing relation between a directory and a file
a linking relation

Figure 6.3. Differences between a directory in the MemorySafe system, a directory in a Unix filesystem, and a directory in a Windows system.

others have exactly one in-coming edge. Figure 6.3 illustrates the differences between the three models.

In Figure 6.3, Diagram A shows an example view of MemorySafe, which is depicted as a connected, acyclic and rooted graph. In Diagram A, any non-root node can be reached from c; a directed edge, say the edge connecting c and f, shows the containing relation between a directory and a file or a directory; g and i have multiple links. They have multiple in-coming edges.

Diagram B of Figure 6.3 shows an example view of a Unix file system. Unlike Diagram A, Diagram B has multi-arrows. In a Unix file system, a user can create a hard link to a file in the directory where the file is stored. So in that directory two filenames refer the same file, as shown by two edges connecting k and j in the diagram. Moreover, unlike with Diagram A, a link is allowed only between a directory node and a file (leaf node) in Diagram B.

Diagram C of Figure 6.3 shows an example view of a Windows file system. Windows systems do not allow linking. Instead, they allow users to create shortcut files. Shortcut files function as links but are treated as files in the systems. In Diagram C, there is at most one edge connecting two vertices, which is different from Diagram A and Diagram B.

Comparison of practical use of different linking mechanisms. Links in MemorySafe are similar to the hard links of the Unix file system. Like hard links in a Unix file system, links in MemorySafe are indistinguishable from the original resource references. Furthermore, every link to a resource in a MemorySafe server must reside on the same server as the resource. However, there are differences between links in MemorySafe and hard links in a Unix filesystem.

- Renaming a photo will automatically change the name of the photo in all resources that have a link to the photo. In a Unix file system, changing the name of a file will have no influence whatsoever on the other links to the file.
- In a Unix file system hard links can only be used for files while in a MemorySafe server links can be used for both files and directories.
- In a MemorySafe server there are concepts that are similar to symbolic (soft) links of a Unix file system.
- In the implementation, the Unix file system keeps track of how many hard links reference a file while the MemorySafe system keeps track of which links reference a file.

In comparison with a Windows file system, both links in a MemorySafe server and shortcut files in the Windows file system can be used for files and directories. However, they have different conceptual models and different implementations. In a Windows file system, a shortcut file contains a reference to a target directory or a target file, like a symbolic (soft) link in a Unix file system. From a user's or a programmer's perspective, the shortcut file is just a normal file in the file system. In MemorySafe, links are indistinguishable from the original resource references. In the Windows file system, cyclic paths can be created using shortcut files. In MemorySafe, no cyclic paths are allowed at all. In the Windows file system, if the target, a file or directory being linked, is renamed, moved or deleted, the shortcut file will become invalid. In MemorySafe there is no such problem.

6.2.4 Server translucency

The MemorySafe system provides server-translucent [38, 142] data access, meaning that a user or a program can choose to work with the system in a server-aware or server-transparent manner. Server-awareness is a must for the development of applications regarding data migration. But as mentioned in Section 6.1.4, digital photos may have copies in different devices, so locating copies can be difficult when using server-aware data access, given a large number of albums and photos. Therefore server-transparent access is preferred wherever possible.

Server-transparency provides a naming mechanism with which a user or a program can access data regardless of where data are stored. Naming is a mapping between logical names and physical resources in the MemorySafe system. Usually a user will refer to a resource by a textual name (pathname). A resource is an abstraction of physical disk blocks on hard disks and it is also an abstraction of replicated data objects.

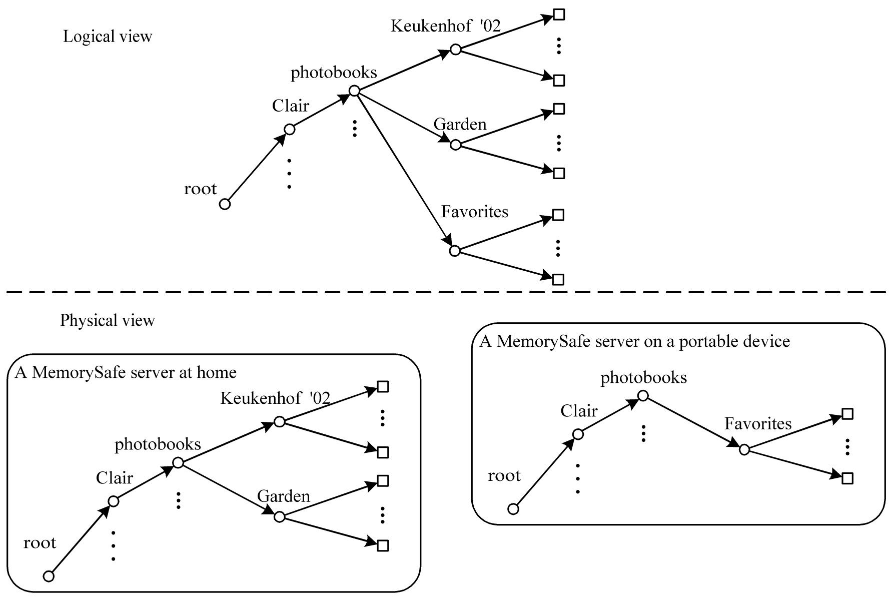


Figure 6.4. The logical view of a user's data in the MemorySafe system is an aggregation of all physical views of a user's data in the MemorySafe servers.

Definition 6.2.1 (Server transparency). *The pathname of a resource does not reveal any hint as to the server in which the resource is located.*[4]

The MemorySafe system fulfills a server-transparent access in terms of server-aware accesses.

Server-transparent accesses. There are two types of data accesses: *read accesses* that do not change the state of the MemorySafe system and *write accesses* that do change the state of the MemorySafe system.

When data are retrieved from the MemorySafe system, the result of a server-transparent read access can be implemented as the aggregation of the results obtained by applying the access to all MemorySafe servers in the

[4] A related concept is *location transparency* [91, 150]: The pathname of a resource does not reveal any hint as to the device in which the resource is stored. The pathname of a resource may tell a user that a resource is located on a certain logical server, but it does not tell the device in which the server is running. Applications using such a pathname will cease to work when the device running the server is disconnected or the requested resource is moved. So location transparency is not a desired property in this context. A concept similar to server transparency is *location independence*: The name of a resource need not be changed when the resource is moved from one server to another. Here, the term "location" refers to the device in which a resource is stored.

system. For example, a user wants to retrieve all his or her personal photo albums in the MemorySafe system, as illustrated in Figure 6.4. The user would expect to retrieve three albums, “Keukenhof ’02” and “Garden” from the MemorySafe server at home and “Favorites” from the server on the portable device. In this case, the information regarding the servers in which data are stored need not be revealed to the user at all.

However, at two different instances of time, a server-transparent read access may have different results. If a user wants to browse his or her photo albums when he or she uses the portable device outside the home, the user will see only the “Favorites” album. This is due to the fact that the portable device is disconnected from the server at home. Where are the missing “Keukenhof ’02” and “Garden” albums? The user’s view on his or her photos is inconsistent before and after the disconnection. This problem is called *data miss.*

A server-transparent write access is even more difficult to implement in dynamic environments, such as home environments, than a server-transparent read access. Let’s continue with the example in Figure 6.4. For instance, the user wants to upload a photo to an album, say “Garden”. To make a best effort to ensure the photo will be available in all the servers, the MemorySafe system would try to upload the photo to all servers holding the album. Suppose that at the time when the operation is performed the portable device is not accessible. The uploading operation will then only add the photo to the album of “Garden” on the MemorySafe server at home. Consequently, the newly added photo won’t be accessible when the user uses the portable device outside the home. This is another case of the data miss problem.

These two examples show the fact that device disconnections may break down server transparency of data accesses and may result in data miss. Therefore, a proper notion that models disconnections should be included in the naming mechanism of the MemorySafe system.

Data locality. Usually people only care about data availability, not necessarily the location where they are stored. But the user does need to know whether certain data will be available or not when his or her portable device is disconnected from the others. This leads to the introduction of *data locality.*

Definition 6.2.2 (Data locality). *A* local *resource is a resource that a user wants to access on his or her portable device, even when the device is disconnected from the others.*

Data locality is integrated into the naming schema of the MemorySafe system. A user can instruct the MemorySafe system to make data in a certain pathname *local*, in other words *locally available.* This operation is called *localize.*[5] The selected data are a small part of the total amount of user data

[5] A related mechanism in distributed file systems is the *mounting mechanism* [91]. A mount operation binds a directory of a file system to a directory of another file

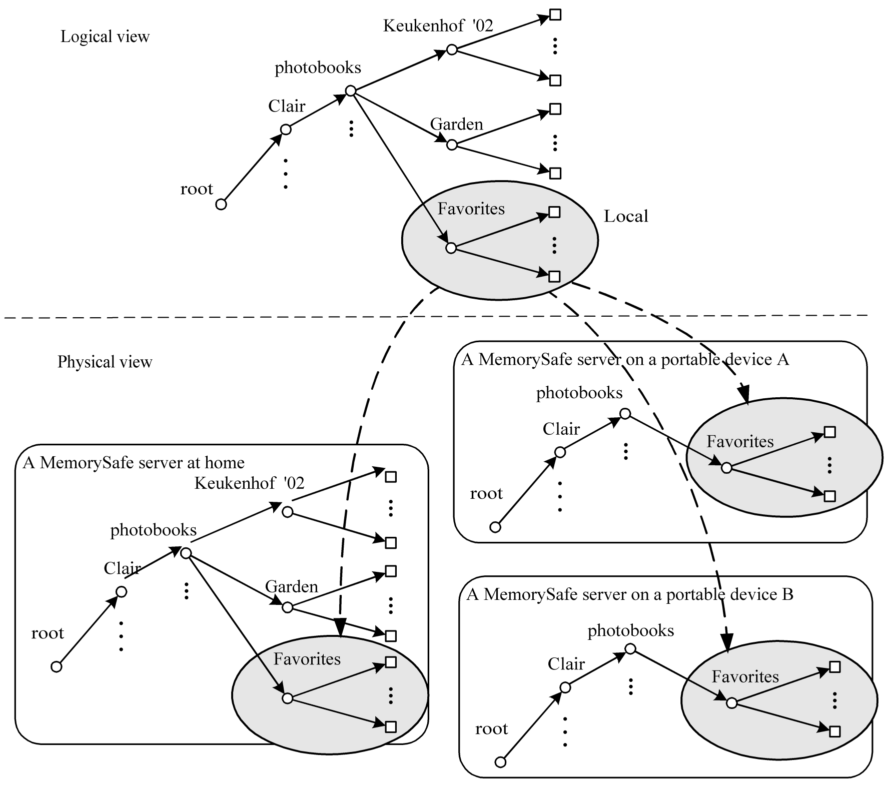


Figure 6.5. In the MemorySafe system, a local album, "Favorites", in the logical view of the system is dynamically mapped to a physical resource in a MemorySafe server, depending on where the access is performed.

and can be transferred to any device that the user will use. Before the device is disconnected, the user will know which data will be available and whether they will fit in a target device. With this information, the user can take measures to avoid data miss.

When a local resource is accessed, the name is dynamically mapped to a physical resource stored on a server in the MemorySafe system. For example, a user makes one of his or her albums, say "Favorites", always locally available, so that he or she can access it no matter which device is in his or her vicinity. This example is illustrated in Figure 6.5. When the user uses the portable device A outside his or her home, the MemorySafe system will map the

system. The formal directory is statically joined to the name space of the latter system. Pathnames of files in the former directory are location-transparent, instead of location-independent. Unlike with the mounting mechanism, pathnames of the resources that are set to be local in the MemorySafe system are dynamically mapped to physical resources in MemorySafe servers and such pathnames are server-transparent.

"Favorites" name to the "Favorites" album stored in device A. Likewise, if the user takes portable device B with him or her and wants to access the "Favorites" album, the name "Favorites" will be dynamically mapped to the "Favorites" album stored in device B. If the MemorySafe system happens to discover that there are several MemorySafe servers that hold an album called "Favorites", the system will aggregate the contents of the "Favorites" albums of these servers and present them to the user.

When a local resource is modified, the modification will immediately be applied to the physical resource stored on the device which the user is currently using. In this way, the user can observe a coherent view on the resource, even when the device works alone. If there are any MemorySafe servers in the vicinity, or when the server is connected to another server, the MemorySafe system will synchronize those modified local resources.

For example, in the scenario depicted in Figure 6.5 the user uploads a photo to the "Favorites" album on portable device A when he or she uses the device away from his or her home. When the device is brought back home, the MemorySafe system will synchronize "Favorites" albums that are stored in different MemorySafe servers, besides aggregating the photos in those albums. In this way, write accesses on local data are propagated from one server to other servers. When the devices become disconnected again, the user will continue to view the aggregated and consistent data.

6.2.5 Identity-based history synchronization

In the MemorySafe system, data that are made locally available by a user will be physically stored in any device in the user's vicinity, as depicted in Figure 6.5. Disconnected updates on such data may lead to data inconsistencies. When the device is connected to another one, the MemorySafe system will synchronize the localized data.

When a user localizes some data, the MemorySafe system will collect localized data from all servers and then replicate all the data to the local server. The localize operation is regarded as an *identity-preserving* operation.

Definition 6.2.3 (Identity-preserving). *The newly introduced resource stored in a device in which a data localization is performed has the same resource identity as the localized resource.*

For example, when the user localizes a photo album, say "Favorites", on a portable device, photos in the "Favorites" album from the other devices will be copied to the portable device in an identity-preserving manner. Later, the preserved resource identity will be used to determine whether two resources have the same origin.

The MemorySafe system uses the identity-based history synchronization of resources. Details can be found in Chapter 7.

6.3 Implementation of the MemorySafe system

6.3.1 The Empire middleware platform

Several middleware platforms were available for programming in dynamic and mobile environments at the time when the Phenom project was started. Jini [145] was the first fully implemented and well-documented platform suitable for fast prototyping and concept proving purposes. A trial version of the MemorySafe system [33] was therefore developed using the Jini technology.

Jini is implemented in Java. Service discovery in Jini is implemented using the Jini-version implementation of the UDP protocol and service communication of Jini is based on Java Remote Methods Invocation (RMI). In the trial system of MemorySafe it was discovered that Jini service discovery was not reliable when devices were connected via wireless Ethernet technologies. It could take several minutes to discover a device joining or leaving and actual time latency was of high uncertainty, which made it almost impossible to give a live demonstration on device discovery for the visitors of the Phenom system. It was also found that RMI-based communications did not perform well in wireless networks. Moreover, Jini did not provide possibilities for fine-tuning to address the unreliability and latency problems.

A commonly used solution to those problems is to overwrite the implementation of Java RMI so that developers can fine-tune reliability and performance of RMI to meet application-specific requirements. For example, caching is used to reduce network traffic over RMI in [39].

In order to obtain total tunability on reliability and performance, the Empire system [30] was designed and implemented by Nick de Jong of Philips Research. Empire is a middleware platform for prototyping applications in home environments.

In the Empire system, devices are interconnected by an Ethernet. Some devices have a wired connection while others have a wireless connection. Empire was fully developed in Java, taking advantage of the platform-independence of the Java technology. In the Empire system, applications running in devices that run the Windows systems, the Unix systems and the Linux systems can communicate seamlessly. The Empire system is light-weight, which promises better tolerance for the heterogeneity of home environments where computing devices have different resource constraints. Like Jini and UPnP, the Empire system provides merely communication mechanisms between applications. It does not have built-in data management modules. The MemorySafe system is a functional component that is responsible for distributed data management.

6.3.2 System architecture

The MemorySafe system consists of a number of MemorySafe servers. Figure 6.6 provides an overview of the architecture of a MemorySafe server. The

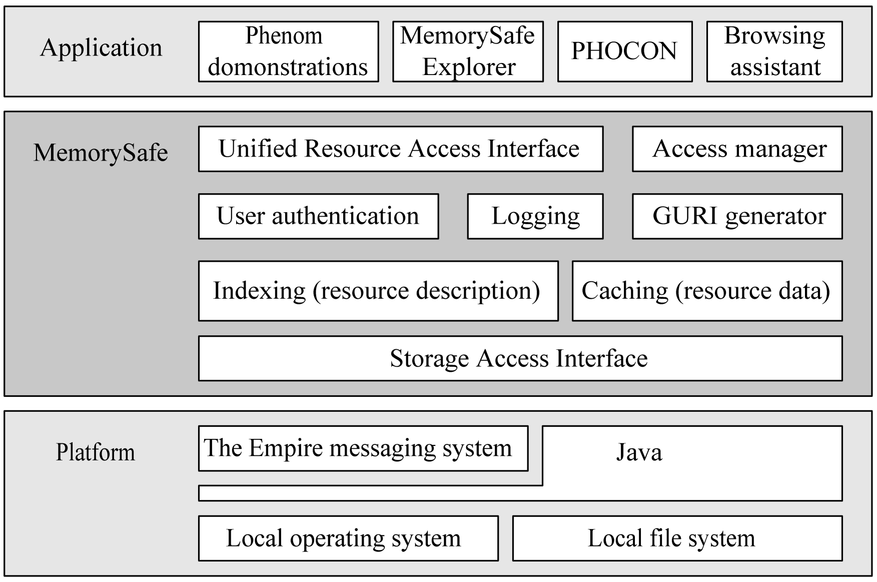


Figure 6.6. An architectural overview of a MemorySafe server.

middle part of the diagram indicates the key mechanisms of a MemorySafe server, among which are *storage access interface*, *indexing* and *caching*.

Storage Access Interface. Each MemorySafe server is a Java object accessing storage via `StorageAccessInterface`, a generic Java interface. The current implementation of the interface provides data access to file systems. Table 6.4 lists the methods that are defined in the interface. In the future, the servers can be implemented using database systems without re-engineering: existing code based on the `StorageAccessInterface` interface need not be changed.

Indexing. A super server may contain a large amount of data. In the current configuration of the MemorySafe system, one super server stores about 5000 resources. The resource description of each resource is stored in a file, called the *info* file, in the local file system. Different resources have different info files. Without indexing, retrieving resource descriptions would introduce frequent hard disk reading operations and become rather slow. To avoid this situation, an index file containing the resource descriptions of all local resources of a MemorySafe server is created.

The index file system is stored in the local file system of a MemorySafe server and loaded into memory when the MemorySafe server is started. Resource lookup and retrieval operations are performed solely in system memory. In this way, the overhead of accessing the hard disk is avoided.

A problem resulting from indexing is the consistency problem. The resource description of a resource is stored in two places: in the info file of the

Table 6.4. Methods defined in the storage access interface.

Category	Methods
authentication	`login`, `addAccount`, `changePassword`, `removeAccount`, `getUsers`, `getCurrentUser`
server management	`isWorking`, `stopService`, `existResourceServer`, `initializeServer`, `format`
pathname-based access	`getByPath`, `getChildrenByPath`, `getParentsByPath`, `getRootGURIs`, `putRootGURIs`
resource lookup	`lookupOnKeyValue`, `lookupOnKeyValueInHierarchy`, `getAllReferencedResource`
per-resource (GURI-based) access	`lookupOnGURI`, `getPaths`, `getChildren`, `getParents`, `createPersistentResource`, `delete`, `copy`, `replicate`, `move`, `isPublic`, `setPublic`, `getResourceMetaData`, `putResourceMetaData`, `getResourceInfo`, `putResourceInfo`, `getResourceData`, `putResourceData`, `getResourceDataVariant`, `putResourceDataVariant`, `getVersionedData`, `getResourceLogData`, `putResourceLogData`, `record`, `flatten`

resource and in the index file. The information in the two places should be identical. When the resource description of a resource is updated, the modification should ideally take place in both places. However, updating the index file is a “heavy” operation. The size of the index file of a super server containing 5000 resources is about 1.1 MB. So updating the index file at real time should be avoided.

To address consistency problems, the indexed information in memory is updated when the info file of the resource is updated, so that the modification will be visible in subsequent resource lookups or retrievals. Instead of updating the indexing file immediately, an update record is appended to a system update log file, called “changes.log”. The next time the MemorySafe server is restarted, a routine will be invoked to apply all the changes recorded in the system update log file to the index file. In this way, the information stored in the index file will become consistent with the information stored in the info files of the resources.

Caching. To improve system performance in handling requests of accessing resource data, each MemorySafe server maintains a local cache in memory. The local cache is used for caching the most frequently accessed resource data and variants, such as thumbnails and images of digital photos. A write-through data cache is used to keep information kept in the cache and stored on hard disks consistent.

6.3.3 Application programming interface

Applications access the MemorySafe system via the unified resource access interface. The interface allows applications to access data in a pathname-based

Table 6.5. Methods for server-transparent access, defined in the unified resource access interface.

Category	Methods
pathname-based access	`getRoots`, `existPath`, `getResourcesByPath`, `getResourceByPath`, `getChildrenByPath`, `getParentsByPath`
per-resource (GURI-based) access	`getParents`, `getChildren`, `getResource`, `create`, `exist`, `setPublic`, `isPublic`, `insert`, `getData`, `getResourceMetaData`, `getNamedData`, `setData`, `setNamedData`, `getDescription`, `setDescription`, `getHistory`, `getPreviousVersion`, `getDescriptor`, `setDescriptor`, `removeDescriptor`, `moveResources`, `getResourcePaths`, `deleteLocalLink`, `deleteLocalLinks`, `deleteLocalResource`, `deleteLocalResources`, `addLinkTo`, `copyResources`, `copyLocalLinks`, `moveLocalLinks`, `cloneResource`, `replicateResource`, `replicateResources`, `getAllHistories`, `getHistory`, `clearHistory`

manner or in a lookup-based manner. Applications can store and retrieve data using pathnames. They can look up resources by specifying criteria. The interface also provides methods for applications to access the MemorySafe system in a server-aware manner or in a server-transparent manner. Table 6.5 lists all the methods that are defined for server transparent data access in the unified resource access interface.

Server-transparency. All MemorySafe servers are organized in groups. Each server group has a unique server group identification, in short SGID, which is provided by the user. All the servers in a server group have the same SGID. Each server group has a super server, acting as a master data repository at home, and a few thin servers, behaving like portable devices. The super server has the name "home" and the thin servers have names other than "home".

Applications that do not involve multiple server groups can be developed without any concern for data distribution. When an application invokes a method of the interface, the server group to which the application belongs will first be automatically determined. If the client device where the application is running does not have a local MemorySafe server, the application will assume that it is part of the "default" server group. If a client has one server running locally, the client will assume that it is part of the group of that server. Next, the method will start three sub-routines in its invocation: (1) Forward the request to the "home" server of the application's server group; (2) Forward the request to the locally running server of the application's server group; (3) Forward the request to the servers of neighbor server groups. Finally, the results of the sub-routines will be merged after the sub-routines have been completed.

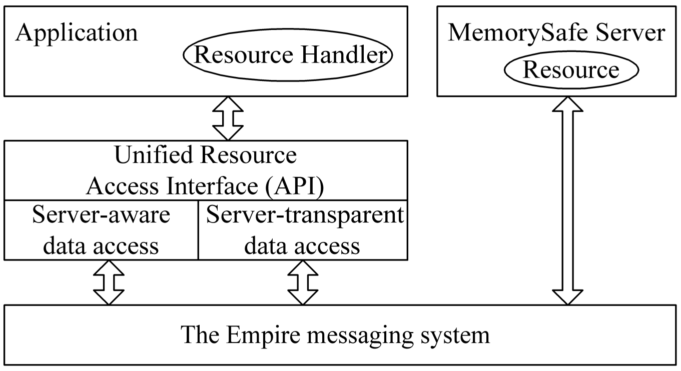

Application
Resource Handler
MemorySafe Server
Resource
Unified Resource Access Interface (API)
Server-aware data access
Server-transparent data access
The Empire messaging system

Figure 6.7. Communication between a client and a MemorySafe server.

Resource handlers. When the MemorySafe system is accessed, applications may manipulate resources residing remotely on MemorySafe servers through *resource handlers*, Java objects residing locally in the Java virtual machine of the client applications, as shown in Figure 6.7. A resource handler caches the resource description and the data of the thumbnail of a resource stored in the MemorySafe system. Caching such data helps to improve GUI rendering by reducing network traffic, which is useful for resources containing images.

In the MemorySafe system, cache consistency of resource handlers is maintained according to application needs.

- By default, resource handlers do not perform any consistency check. This is based on the following considerations. In the first place, using any server-oriented cache invalidation approach, the MemorySafe servers of the MemorySafe system would have to keep track of client applications. This adds overhead on the portable devices with a running MemorySafe server, which has resource constraints. Secondly, the MemorySafe system is designed for environments in which devices can join and leave anytime. Tracing device activities adds overhead for devices hosting MemorySafe servers. Thirdly, in the MemorySafe system, resources that store digital photos are seldom modified. Constant consistency checks at client applications would affect the performance of photo-browsing related applications.
- Alternatively, applications can turn on consistency check switches of resource handlers, which will verify the value of the last-modification property of a resource handler against that of the resource residing in the MemorySafe system and refresh cached data in the case of any inconsistencies.

In addition, resource handlers are capable of disconnection handling. Applications access the MemorySafe system by using pathnames and they receive resource handlers in return. Resource handlers maintain connections with the MemorySafe system via Empire. If the MemorySafe server which

Table 6.6. Statistics of the Java source code of the MemorySafe system.

Package	Classes	Lines of code	Size of file (KB)
Core code	23	9733	307
Communication	6	6059	231
Utilities	14	3944	116
GUI tools *	8	4142	143
Total	51	23878	797

* This includes the MemorySafe Explorer, a GUI tool for managing resources, and does not include the GUIs of the Phenom project.

serves a request from a client application becomes inaccessible due to device failure, shutdown or disconnection, the resource handler held by the client application can automatically start to search for a running server which can fulfill requests of the client application. Only if the resource handler can't find any backup server will an error message be reported to the client application.

6.3.4 Coding and use of the MemorySafe system

The MemorySafe system was fully implemented in Java. Table 6.6 gives some statistics of the system's Java source code. MemorySafe was installed in the Phenom development room on April 9, 2002. The system has been running since its installation. On May 21, 2003, it was recorded that 4778 resources were stored in the system and that the system had a total of 1.01 GB data stored in hard disks. Resource descriptions, update logs, versions of data and data variants are stored for each resource. In the system there are different types of data: images, audio clips, settings and plain text data. Several applications have been built on top of the MemorySafe system. They will be introduced in Chapter 8.

6.4 Related work

Many distributed systems have been designed to address disconnected updates, as discussed in Section 1.6.2. Those systems were designed on top of existing file systems or database systems. Those systems do not tackle multiple copies of multimedia data. Moreover, they were built for office environments, regardless of requirements such as heterogeneity in home environments. Therefore, those systems, strictly speaking, don't provide the openness and interoperability which are required to enable such systems to be deployed in home environments.

In addition, the MemorySafe system deals with data hoarding differently from those systems. AFS [65, 66] uses local caches to hoard files in a transparent way. The user knows the recent access files are available, but does not know what files they are. Hoard profiles [79] are used to determine the files that should be replicated on client devices. Coda uses the same approach [74]. In Coda, such hoard profiles are called hoard databases, which are per-client. Moreover, Coda uses the snapshot spying method, in which the user sets up bookends to delimit a period of activity [132]. All files accessed during this period are to be hoarded. The semantic distance approach [80, 81] calculates the time, the number of close operations between two open operations to determine the files to be hoarded. The transparent analytical spying method [148] detects working sets for applications and data, provides generalized bookends to delimit periods of activity, and allows users to load a briefcase. In disconnection database systems, hoard attributes [111] are used to partition data sets. Generally speaking, hoard attributes are attributes capturing locality of access. DOC [62] and PFS [37] simply use the cached data as hoarding data. In these systems, users are not fully aware of which files are replicated to local storage devices. In the MemorySafe system, the localize operation allows users to explicitly define data to be hoarded on mobile devices. In this way, users have full control over hoarding data.

The MemorySafe system has several features in common with Roma [146]. Both allow fully-extensible attributes for metadata of data objects. Both help applications maintain the connection between logically related copies of a file by assigning a unique identifier that is common to all of the copies.

The MemorySafe system also resembles the semantic file system [49]. Both allow applications and users to provide rich searchable metadata on resources. However, the MemorySafe system has a distributed architecture supporting disconnected updates, aggregate logically related variants and several other features that do not appear in the semantic file system.

There are also several middleware platforms for developing multimedia-related applications. TOAST [46] is a middleware platform providing CORBA support for adaptive distributed multimedia applications. TOAST was developed mainly for distributed cooperative visualization and handles disconnected updates. COTS [73] provides dynamic connection of commercial-off-the-shelf components (hardware and software components) on demand, supporting ad-hoc interaction. It does not address data replication at all. A resource replication model [138] is proposed for network information appliances to collect and manage data in middleware platforms. The model exploits Java technology and XML [106] to construct data replication services. An open architecture is proposed for the integration of data and media objects in an object-orientated database [20]. This platform does not take into account multimedia-specific requirements, such as object copies. The concept of resources is also mentioned in the resource replication model [138]. It is not clear whether the resources mentioned in [138] have elements similar to the access history or data variables of MemorySafe. The concept of resources is similar to that of "items" in Unidata [4]. In Unidata, the contents are grouped into logical and atomic units, distributed via an information channel from system resources to a set of users. It is used in publish/subscribe systems in mobile information systems.

ShoeBox [101, 128] is a digital photo management system developed at AT&T Research Laboratories at Cambridge, UK. This system is designed merely for photo storage and retrieval, especially integrated with speech recognition for photo annotation and retrieval. Unlike ShoeBox, MemorySafe can be used for other multimedia data besides digital photographs. MemorySafe can also be used for storing working documents and application data. In the ShoeBox system, photos are treated as atomic data objects stored in an object-oriented database. In ShoeBox, copies of a photo are treated as separate data objects, different from the original photo. In MemorySafe, logically closed related data objects are aggregated into resources. Each resource has a unique identity and is capable of storage and retrieval of different copies of a photo. In ShoeBox, albums can be defined. However, no further hierarchies are allowed. Unlike ShoeBox, MemorySafe allows flexible data structures, namely acyclic directed connected graphs, to be constructed whenever desired. ShoeBox has a centralized architecture while MemorySafe is a distributed system, providing data replication and data synchronization.

6.5 Concluding remarks

Designing a distributed data management system for home environments is not just for computer scientists and engineers. It should also involve experts on user-system interactions and professionals in the field of home networking technologies. The MemorySafe system is the result of a research attempt to develop in-home systems in an industrial setup.

The MemorySafe system has so far been used mainly in the Phenom project. The system has not yet been used for application development in other Philips projects. It has not yet been used by people outside the Phenom project either. The main reason for this is that the current implementation of the MemorySafe system relies on the Empire messaging system. Nevertheless, concepts, ideas and solutions that originated in the MemorySafe system have been discussed with researchers in other Philips departments during guest visits and corporate research exhibitions. Some have since then been used in other projects of Philips Research. The MemorySafe system was also presented and demonstrated at the International ITEA Workshop on Virtual Home Environments [121].

7. Data synchronization in the MemorySafe system

In the MemorySafe system, characteristic-entry logs are used to record disconnected updates and identity-based history synchronization is used to resolve data inconsistencies. This chapter describes how data synchronization is implemented in the MemorySafe system. Moreover, an empirical study on the use of characteristic-entry logs will be described.

7.1 Access histories

To support identity-based history synchronization, each resource in the MemorySafe system has an access history, which records every modification on the data or description. Each entry in the access history has five fields storing different information: timestamp, operation, descriptor, pre-value and post-value. The values of the fields descriptor, pre-value and post-value are determined by applications. The timestamp field is filled in by the system. The operation field depends on the action performed. Its value is either `set` or `remove`.

For the resource description of a resource, every insertion, updating and removal of a descriptor is logged. For instance, the addition of a descriptor to the description of a resource is recorded in the access history as follows.

```
<1032359455345, set, location, null, Keukenhof>
```

The first element is a timestamp, the second is the type of operation, the third is the key of the modified descriptor, the fourth is the value of the key before the modification, and the last element is the value of the key after the modification. The pre-value of this descriptor, `null`, indicates an insertion of the descriptor.

The name of a resource is treated as a descriptor of the resource. The renaming of a resource is recorded in the access history as follows.

```
<1032359466080, set, name, anonymous, flowers>
```

The removal of a descriptor is recorded as follows. After the removal, the key of the descriptor becomes invalid. The post-value of this descriptor, `null`, indicates this invalidness.

```
<1032359498080, remove, title, Flower pictures, null>
```

When the data part of a resource is modified, an identification is assigned to the new data version and the identification is stored in the resource description. So updating the data of a resource is treated as an update on the `latest_version` descriptor of the resource. In the access history, a log entry recording the modification of `latest_version` is added.

```
<1045227651908, set, latest_version, null, 1045227651818>
```

Adding or modifying a data variant of a resource, for instance making a thumbnail for the resource, introduces a new version of the data variant. In the access history, a log entry recording the modification of the data variant is added. The unique identification of the new version of the data variant is used.

```
<1045227652559, set, latest_version_thumbnail, null, 1045227652539>
```

7.2 Implementation of access histories

In the MemorySafe system, access histories record modifications on resources. Several major design decisions were taken in the actual implementation of access histories.

Normal logs as access histories. Access histories were initially implemented as characteristic-entry logs. In an early evaluation of the system in the context of the Phenom project it was agreed that users of this system would presumably be in favor of the feature that whenever needed, the system should provide all updates that a user performed on certain resources. Since characteristic-entry logs do not offer such traceability, it was decided to implement access histories as normal logs.[1]

Therefore, access histories of resources were finally implemented as normal logs. Access histories are real-time converted to characteristic-entry logs when they are exchanged between devices during data synchronization. The correctness of using normal logs and characteristic-entry logs at the same time in data synchronization, proven in Theorems 5.3.2, 5.3.4 and 5.3.6, provides a formal justification for this treatment.

An additional advantage of using normal logs is that logs without truncation provide authentic data for evaluating the use of characteristic-entry logs, if they were implemented. One case study of this sort will be described in Section 7.4.

[1] When access histories are exchanged during data synchronization, access histories are real-time converted to characteristic-entry logs in the implementation.

No reads. The MemorySafe system was designed for supporting photo-browsing applications of the Phenom project. The applications developed in the project involve heavy screen rendering operations, especially for thumbnails of digital photos. Performance of the MemorySafe system in feeding data is hence crucial to those applications. Since logging all read operations on data would otherwise introduce overhead of flushing logging information onto hard disks, it was decided that read operations should not be logged in the implementation of access histories.[2]

Simplification. In the MemorySafe system, logging updates does not involve the application's awareness. When specifying a new descriptor of a resource, the application would typically first add the descriptor to the resource and then assign a new value to the descriptor. A simpler way of doing this is to allow the application to specify the initial value when adding the descriptor to the resource, which is actually provided by the application development interface of the MemorySafe system.

One execution of this high-level operation would involve two log entries appended to the access histories of the resource. According to the model developed in Chapter 3, one entry is for the addition of a new descriptor and another is for setting an initial value of the descriptor. When examining any access history, the user would expect the MemorySafe system to show only high-level operations, instead of detailed low-level log operations. To solve this mismatch, the final implementation of access histories deviated slightly from its original design. Adding a descriptor with an initial value to a resource is treated as a `set` operation. When this write operation is logged, the pre-value of the descriptor is left “null” in the log entry.[3]

Encapsulation of access histories with resources. Considering that data are subject to migration in the MemorySafe system, access histories of resources are stored in separate files in file systems of the devices running MemorySafe servers. Data modifications are not stored in a single file

[2] In the model of normal logs developed in Section 3.3, four types of operations, notably add, read, write and delete, were included to model a variety of system behaviors for richness and completeness. In the MemorySafe system ignoring read operations is a simplification of that model. This treatment is justified by the fact that several semantic rules such as “weak no-update-loss” do not require information on read accesses, as shown in Sections 5.2 and 5.3. Ignoring read accesses will hence not lose valuable information necessary for data synchronization in the MemorySafe system when those rules are applied.

[3] The add and write operations are treated in the same way when semantic rules such as “no-update-loss” and “weak no-update-loss” are used in data synchronization, as shown in Sections 5.2 and 5.3. It will hence not affect the result of data synchronization when adding a descriptor with an initial value to a resource is treated as a “set” operation. Therefore the “set” operation appearing in log entries of access histories in the MemorySafe system works always like the “write” operation in the formal model of logs developed in Section 3.3, except that a log entry with a “set” operation indicates an addition of the descriptor in the case of the pre-value of the specified descriptor being “null”.

on each MemorySafe server. This treatment simplifies preparation for data migration. Collecting all meta-information of a resource to be migrated does not require any filtering of relevant log entries from a lengthy log file, which would otherwise be necessary if access histories of resources had been stored in a single log file. Moreover, this treatment makes it easier to truncate an access history of a resource to a characteristic-entry log.

Directories as sequences of resource references. One distinct use of directory resources in the MemorySafe system involves their use as photo albums. MemorySafe allows users to define the ordering of photos according to their own interests. In addition to ordering by name or shooting date, a user can manually arrange images on the basis of contents or locations. The defined ordering is then used in displaying photos in a slide-show mode, for example. So preserving the ordering information of contained references is a required feature.

In the MemorySafe system, directory resources are treated in the same way as normal file resources. The data part of a directory resource is a sequence of ordered references to other resources. Any modification on this sequence, such as adding, removing or ordering references, is treated as a modification to the data part of the resource. It introduces a new data version and the identification of the new version is saved in the resource description of the directory resource.

Thanks to this treatment, users of the MemorySafe system can also specify thumbnails of directory resources containing digital photos, for the purpose of album browsing. The provided thumbnail of a directory resource can be stored as a data variant of the resource. Additional metadata can also be added to the description of the resource.

7.3 Using access histories in data synchronization

Data synchronization in the MemorySafe system is invoked in a server-transparent access or is instructed by a user via provided synchronization tools. In data synchronization of the MemorySafe system, access histories are first truncated to characteristic-entry logs. Next, characteristic entries are exchanged between devices. After that, collected characteristic entries are serialized in the device in which the synchronization was initiated. Finally, serialized log entries are propagated to all devices participating in the synchronization and applied to the data in those devices.

File resources. In the MemorySafe system, the introduction of GURIs helps to solve the identity loss problem and resolve name-related conflicts, which would otherwise occur in pathname-based synchronization. In the MemorySafe system, most relevant conflicts to be resolved are set/set conflicts and set/remove conflicts.

- Set/Set conflicts. A descriptor has different values in different devices.
- Set/Remove conflicts. A descriptor is assigned with a new value in one device while the same descriptor is removed from another device.

These two types of conflicts correspond to update/update conflicts and update/delete conflicts, respectively, which were discussed and illustrated in Table 2.3.

By serializing resource histories, a set/set conflict can be resolved by applying the latest "set" operation on both sides. In handling a set/remove conflict, the weak no-update-loss rule is applied. With this rule, a remove operation is propagated only if it was performed in the same device in which the most recent set operation was performed. The fact that the most recent set and remove operations were performed in the same device gives a strong indication that this device is the one the user has most recently used. Those modifications occurred in that device are hence considered to be close to the user's intention and should be applied.

In handling modifications on the `latest_version` descriptor, the MemorySafe system will ensure the right version of the data to be propagated, in addition to maintaining the same identification to the version in the resource description.

Directory resources. When resolving inconsistencies of directory resources, special attention must be paid to the modification of the data part of the resources. An inconsistency between the data part of two copies of a directory resource means that the copies will not have an identical sequence of references. This might be due to either of the following two situations.

- One sequence contains a reference that the other one does not have.
- Two resource references do not preserve the same ordering in the two sequences.

To resolve data inconsistencies between two directory resources, the resource references, i.e. the GURIs of the element resources, are merged and propagated to both sides. If there is any reference to a resource that does not appear on the other synchronization side, the resource will be automatically replicated. If an element resource already exists on both sides, the synchronization will proceed at the level of the element resources.

To maintain the ordering of the element resources, the MemorySafe system takes the ordering of resources in the device in which the synchronization was initiated as the primary ordering. Additional resource references from other devices are added to the end of the primary ordering. After this process, the merged and well-ordered sequence of resource references is propagated.

7.4 Empirical study of access histories

An empirical study was carried out to validate the design of characteristic-entry logs. In the MemorySafe system, each resource has an access history recording all the modifications on the resource. This treatment provided an opportunity to examine the actual use of characteristic-entry logs.

7.4.1 Overview

The MemorySafe system recorded all modifications on resources since its installation on April 9, 2002.[4] On May 21, 2003, a snapshot of the MemorySafe system was taken for system analysis. In total, 86607 updates on 4778 resources had been recorded in 408 days. Those updates were stored in files and the sum of the size of those files was 25.7 MB. On average, there were 312 bytes per log entry.

Figure 7.1 shows the number of updates per day in the analysis period. Notably, on April 9 and 10, 2002, there were a large number of updates, as can be seen in the figure. Those updates were introduced because after the installation of the MemorySafe system, data that had been stored in a temporary data registry were imported into the MemorySafe system. The figure also shows that the MemorySafe system was updated far more frequently in March, April and May 2003 than in other months. This is because the Phenom project was closely involved in the preparation of the "Memory Sharing" demonstration for the Philips Research Exhibition 2003 in that period. Between the installation of the MemorySafe system and the preparation of the demonstration, the MemorySafe system was not modified very often. Table 7.1 provides a statistical categorization of the days when the updates were made in terms of the number of updates per day.

Table 7.2 provides an overview of updates per resource type and data type. As can be seen in this table, 86.94% resources were "jpg" resources, recording digital photos. They accounted for 81.90% of the total number of updates that were made in the period of data sampling. This is due to the fact that the MemorySafe system was used largely for storing digital photos in the Phenom project. The second largest portion of updates were updates on directory resources. There were 354 directory resources, which

[4] It is worth mentioning that the MemorySafe system experienced log removal once in the period of this case study. It was found that a resource recording the setting information of the application, Memory Sharing, had been modified more than 14000 times in a month. This was due to heavy experimentation and fine tuning of the user interface of the Phenom "Memory Sharing" application. Most of the modifications were related to temporary adjustments of the user interface. They did not have any further use. At the request of the application's developer, all updates of that resource were removed from the system. The updates contained in the access history of that resource were modifications made after the removal. There was no further removal or addition of logs in the MemorySafe system and the sampling data for this case study was archived.

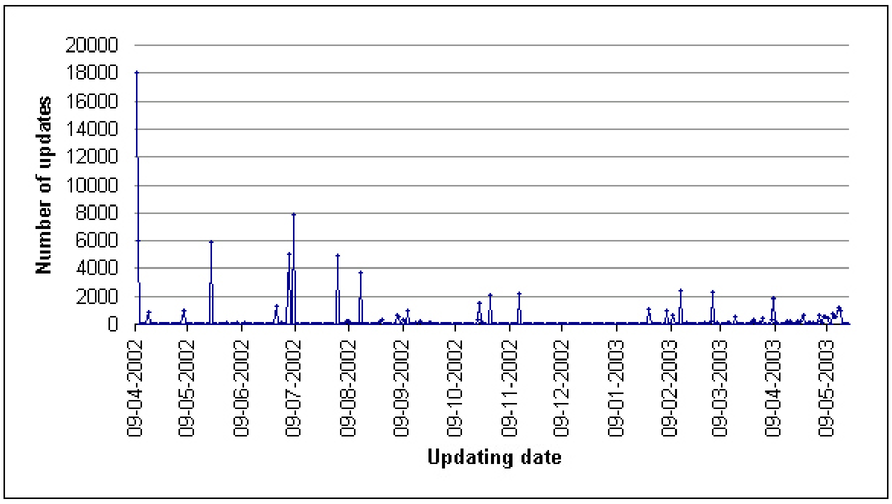


Figure 7.1. Updates that were made on the MemorySafe system per day, in the period from April 9, 2002 until May 21, 2003.

Table 7.1. Categorization of the days (408 in total) when the updates were made in terms of the number of updates per day.

Number of updates per day	Days	Days in percentage (%)
0	279	69.38
$> 0, \leq 100$	68	16.67
$> 100, \leq 200$	19	4.66
$> 200, \leq 500$	11	2.70
$> 500, \leq 1000$	15	3.68
$> 1000, \leq 2000$	5	1.23
$> 2000, \leq 5000$	5	1.47
$> 5000, \leq 10000$	4	0.98
$\geqslant 10000$	1	0.25

were updated in total 9394 times in the analysis period. The other resources together accounted for less than 10% of the total number of resources in the system and updates on them were less than 10% of the total number of updates in the analysis period.

Notably, directory resources, “emp” resources and “txt” resources were updated more frequently than the other resources in the MemorySafe system in terms of updates per resource or updates per resource and day. As can be seen in Table 7.2, the average number of updates per resource, updates per day and updates per resource and day of these three types of resources were above average. This could be attributable to several system activities. In the first place, importing digital photos into the MemorySafe system involves frequent updates on directory resources. Secondly, “emp” and “txt” resources

Table 7.2. Statistic analysis of updates that were made on the resources of the MemorySafe system in the period from April 9, 2002 until May 21, 2003, 408 days in total.

Resource type	Data type	Updates		Resources		Average		
		#[a]	%[b]	#[c]	%[d]	UPR[e]	UPD[f]	UPRD[g]
dir[h]		9394	10.85	354	7.41	26.54	23.03	0.0650
nondir[i]	jpg	70928	81.90	4154	86.94	17.07	173.84	0.0418
	emp[j]	2481	2.86	93	1.95	26.68	6.08	0.0654
	txt	2023	2.34	53	1.11	38.17	4.96	0.0936
	gif	1470	1.70	104	2.18	14.13	3.60	0.0346
	mp3	144	0.17	9	0.19	16.00	0.35	0.0392
	psd	98	0.11	7	0.15	14.00	0.24	0.0343
	cfg	34	0.04	2	0.04	17.00	0.08	0.0417
	wav	34	0.04	2	0.04	17.00	0.08	0.0417
Total		86606		4778		18.13	212.27	0.0444

[a] Number of updates.
[b] Percentage of updates on a resource type.
[c] Number of resources.
[d] Percentage of resources of a resource type.
[e] Updates per resource.
[f] Updates per day.
[g] Updates per resource and day.
[h] Directory resource.
[i] Non-directory resource.
[j] The data type of resources was left empty by applications.

store application configuration, setting and status information, which are frequently modified during application development or are often updated periodically to keep certain application parameters up-to-date.

7.4.2 Performance

To examine the use of logs, the analysis was carried out at two levels. First, the overall performance of characteristic-entry logs was examined, given the sampling data collected in the analysis period. Next, the performance analysis was carried out for each resource type to check whether using characteristic-entry logs would result in any performance improvement. More specifically, directory resources, “jpg” resources, “emp” resources and “txt” resources were considered in this analysis. The other types of resources, such as resources containing MP3 or WAV data, were not considered, since they did not occur significantly in the MemorySafe system.

System performance. Figure 7.2 shows the overall performance of using characteristic-entry logs, given the modifications collected in the analysis

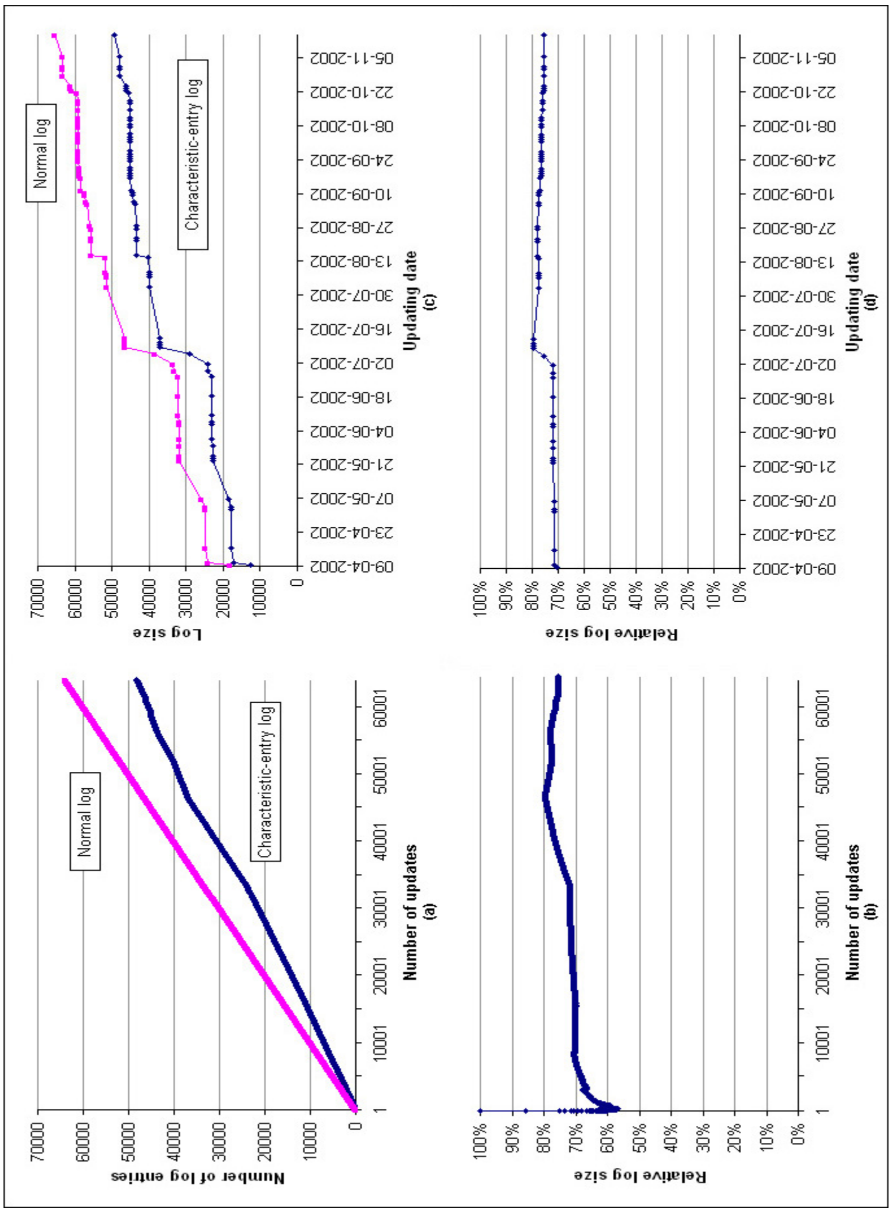


Figure 7.2. System performance when using characteristic-entry logs. Figure (a) shows how the size (the number of log entries) of a characteristic-entry log and that of a normal log change as the number of updates increases. Figure (b) shows how the relative size of a characteristic-entry log (the size of a characteristic-entry log divided by the size of a normal log) changes as the number of updates increases. Figure (c) shows how the size of a characteristic-entry log and that of a normal log change over time. Figure (d) shows how the relative log size of a characteristic-entry log changes over time.

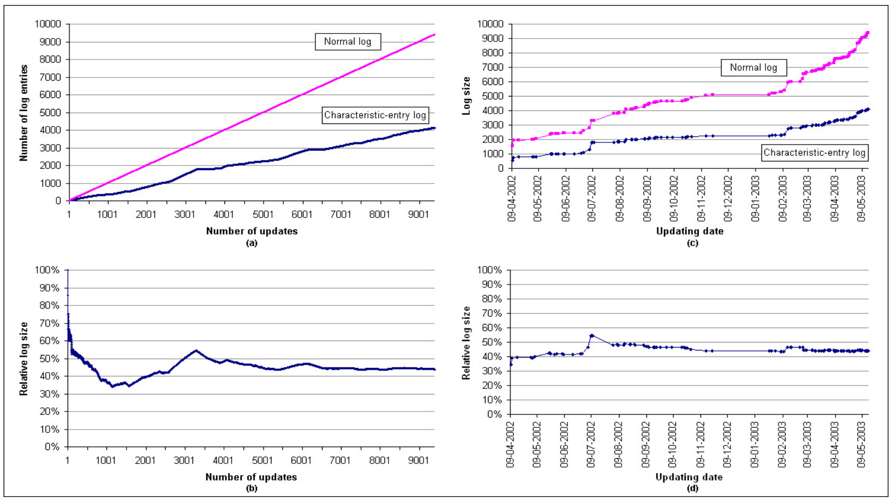


Figure 7.3. System performance when using characteristic-entry logs for directory resources. Figure (a) shows how the size of a characteristic-entry log and that of a normal log change as the number of updates increases. Figure (b) shows how the relative size of a characteristic-entry log changes as the number of updates increases. Figure (c) shows how the size of a characteristic-entry log and that of a normal log change over time. Figure (d) shows how the relative size of a characteristic-entry log changes over time.

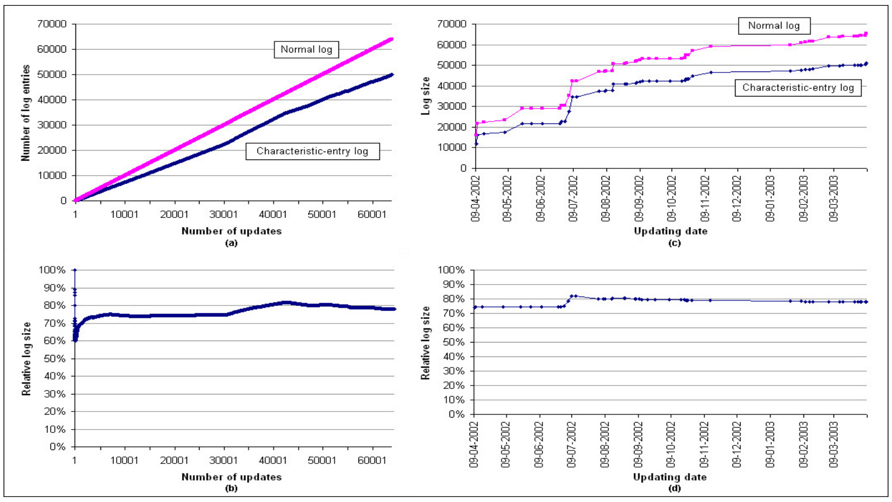


Figure 7.4. System performance when using characteristic-entry logs for "jpg" resources. Figure (a) shows how the size of a characteristic-entry log and that of a normal log change as the number of updates increases. Figure (b) shows how the relative size of a characteristic-entry log changes as the number of updates increases. Figure (c) shows how the size of a characteristic-entry log and that of a normal log change over time. Figure (d) shows how the relative size of a characteristic-entry log changes over time.

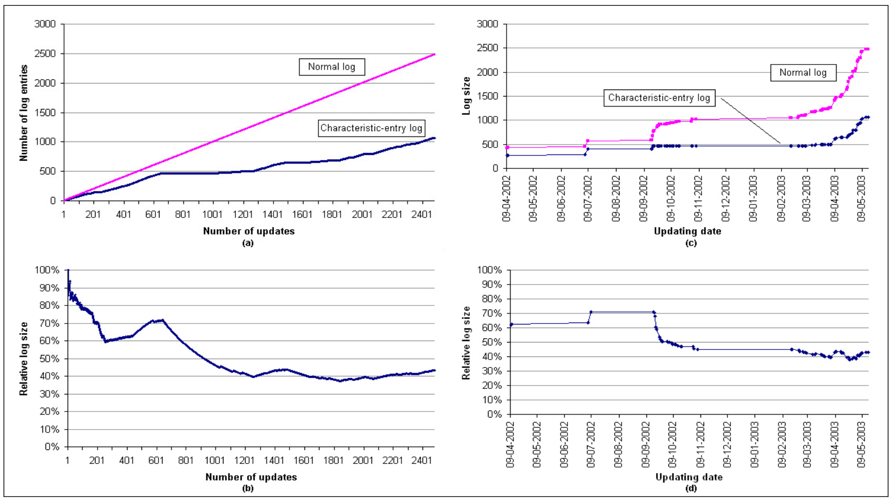


Figure 7.5. System performance when using characteristic-entry logs for "emp" resources. Figure (a) shows how the size of a characteristic-entry log and that of a normal log change as the number of updates increases. Figure (b) shows how the relative size of a characteristic-entry log changes as the number of updates increases. Figure (c) shows how the size of a characteristic-entry log and that of a normal log change over time. Figure (d) shows how the relative size of a characteristic-entry log changes over time.

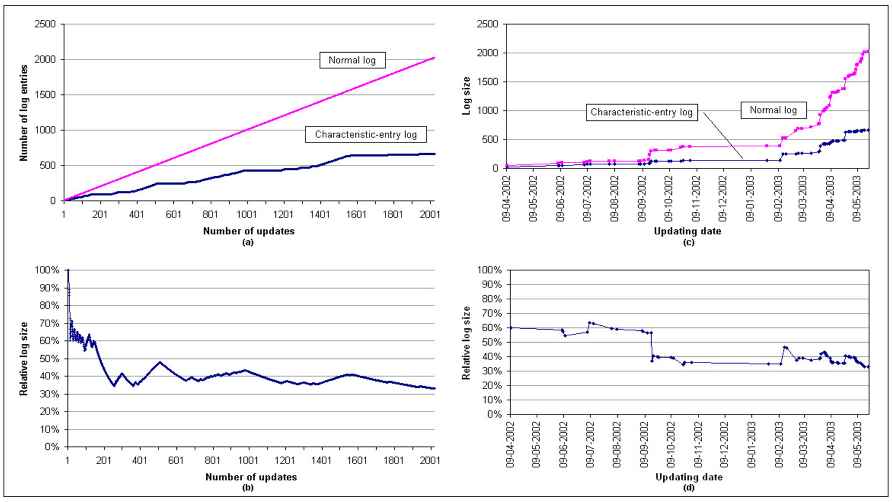


Figure 7.6. System performance when using characteristic-entry logs for "txt" resources. Figure (a) shows how the size of a characteristic-entry log and that of a normal log change as the number of updates increases. Figure (b) shows how the relative size of a characteristic-entry log changes as the number of updates increases. Figure (c) shows how the size of a characteristic-entry log and that of a normal log change over time. Figure (d) shows how the relative size of a characteristic-entry log changes over time.

period.[5] Figure 7.2 (a) illustrates how the log size, i.e. is the number of log entries, of a normal log and that of a characteristic-entry log changed as the number of updates increased. As expected, both log sizes increased linearly with the number of updates and the size of a characteristic-entry log increased relatively slowly.

Figure 7.2 (b) shows how the relative log size, i.e. the size of a characteristic-entry log divided by the size of a normal log, changed as the number of updates increased. The larger the relative log size, the less resource is saved when a characteristic-entry log is used. Less resource use means less memory or storage use. Moreover, it implies less network traffic in transferring log entries over the network during data synchronization and, thus, less time is needed for data synchronization. In Figure 7.2 (b) the relative log size initially drops to about 56% and gradually increases to and remains at about 75%. Using characteristic-entry logs would save 25% system resource in this case study.

In Figure 7.2 the diagrams (c) and (d) show how the sizes of a normal log and a characteristic-entry log and the relative size of a characteristic-entry log, respectively, change over time. The relative log size illustrated in Figure 7.2 (d) confirms the overall gain in performance that can be achieved by using characteristic-entry logs as revealed by Figure 7.2 (b).

Performance per resource type. Figures 7.3, 7.4, 7.5, and 7.6 illustrate the system performance of using characteristic-entry logs for directory resources, “jpg” resources, “emp” resources and “txt” resources, respectively.[6]

Figure 7.3 shows that using characteristic-entry logs for directory resources would consume about 45% of the resources that were needed for normal logs and, thus, save about 55% of the system resources. Such improvement might not be significant for massive storage devices. For miniaturized portable devices with resource constraints, using characteristic-entry logs means less storage, less network traffic for propagating updates, and less power consumption.

In the case of “jpg” resources, using characteristic-entry logs would save about 22% of the system resources, as shown in Figure 7.4. This figure is close to the performance improvement realized by using characteristic-entry logs for the whole system. The main reason for this is that the majority of resources in the MemorySafe system were “jpg” resources and the overall system performance of using characteristic-entry logs was hence largely influenced by “jpg” resources. Regardless of the cumulative effects caused by “jpg” resources, using characteristic-entry logs for directory resources, “emp”

[5] Due to the limitations of the analysis tools that were used in the case study, only the first 65535 updates of the collected 86606 updates were used to draw the diagrams in Figure 7.2.

[6] Due to the limitations of the analysis tools used in this case study, only the first 65535 updates of the collected 70928 modifications on “jpg” resources were used in drawing the diagrams in Figure 7.4.

Table 7.3. Effectiveness of truncating a normal log to a characteristic-entry log.

Resource type	Data type	Days[a]	NLog[b]		CEL[c]		CELog/NLog
			Entries	EPD[d]	Entries	EPD	%
dir		103	9394	91	4117	40	44
nondir	jpg	65	70928	1091	55123	848	78
	emp	69	2481	36	1070	16	43
	txt	65	2023	31	665	10	33
	gif	3	1470	490	1248	416	85
	mp3	3	144	48	108	36	75
	psd	1	98	98	84	84	86
	cfg	1	34	34	24	24	71
	wav	1	34	34	24	24	71
Total			86606		62463		72

[a] Actual number of days when resources were updated.
[b] Normal log.
[c] Characteristic-entry log.
[d] Log entries per day.

resources and "txt" resources would greatly improve system performance, as can be seen in Figures 7.3, 7.5 and 7.6.

As for "emp" resources, using characteristic-entry logs would consume about 45% of the resources that were needed for normal logs and save 55% of the system resources, which is shown in Figure 7.5. In the case of "txt" resources, using characteristic-entry logs would consume about 35% of the resources that were needed for normal logs and save 66% of the system resources, as can be seen in Figure 7.6.

7.4.3 Summary

In this case study several observations were made on the MemorySafe system. The findings can be summarized as follows.

- In a MemorySafe-like system storing personal digital assets data are not updated many times per day on average. Occasionally, the system will experience a large number of updates in one day, due to managerial activities such as image importing.
- Directories are updated more frequently than files. This is largely due to data migration within and/or between devices.
- Digital photographs are not updated as frequently as text data.
- Application-specific data, such as user profiles, are updated more frequently than image-like digital assets.

Table 7.3 illustrates the performance of characteristic-entry logs for the different resource types considered in this study. Characteristic-entry logs perform better than normal logs for recording updates on directory resources,

resources storing application-specific data and text resources. They consume less than 50% of the total amount of storage that would otherwise be needed for normal logs.

7.5 Concluding remarks

For future miniaturized devices such as pocket computers, mobile phones and PDAs, characteristic-entry logs will be rather useful. Those devices have slower CPU and less storage capacity than stationary desktop or laptop computers. Recording and communicating full normal logs would otherwise slow down the synchronization process and overuse the limited power. Using characteristic-entry logs is an efficient way of recording system updates and exchanging updates.

Characteristic-entry logs need further proof in practical use. In the first place, the data that have been analyzed were largely the results of professional activities and may do not really show how people actually use their own devices and systems in their daily lives. Secondly, the case study was performed in the Phenom's development room, examining the use of the MemorySafe system only. The frequency and intensity of the system's use do not necessarily reflect the actual use of other devices and systems in home environments. Thirdly, this study covers mainly image data, text data and application-specific data. Other sources of data such as music and personal documents have not been investigated. This is because the project in which the research was conducted focused only on digital photographs. Once experience has been gained in extensive use of the MemorySafe system it will be possible to perform similar studies to find out how people would access other sorts of multimedia data.

8. Applications of the MemorySafe system

Several applications have been built on top of the MemorySafe system. In this chapter it will be shown how the MemorySafe system supports those applications. Those applications serve to validate the design of the MemorySafe system and to show how extra functionality can be added to the system.

8.1 The MemorySafe Explorer

The MemorySafe Explorer is a user-level tool designed by the author for managing data in the MemorySafe system. A snapshot of the user interface of the tool is shown in Figure 8.1.

Server-translucent data access. The MemorySafe Explorer offers server-translucent data access, which means that a user can choose to work with his or her data in the server-aware access mode or in the server-transparent access mode. In the server-aware access mode, the tool allows a user to manage data stored in individual devices. It offers the following functionalities.

- User account management.By default, a guest account is created. Using this account, a user can create a personal account. In the MemorySafe system, only an authorized user of a MemorySafe server can add a new user account to the server. An authorized user of a MemorySafe server can change the password of his or her account and remove his or her account from the server at his or her will.
- Resource management. In a MemorySafe server, a user has his or her own root directory, in which he or she can store data. Once a new user has been added to a MemorySafe server, a root directory is created for the user. After logging onto the server, the user can manage the data in his or her own root directory. The user can upload photos from local file systems to a MemorySafe server. He or she can modify the metadata and create different copies of a given photo. The user can selectively choose photos from different albums to compose a new album, manually order photos in each album and move data from server to server using the operations provided by the MemorySafe Explorer.
- Resource sharing. A user can share personal data with other users in the MemorySafe system by changing the publicity property of his or her data.

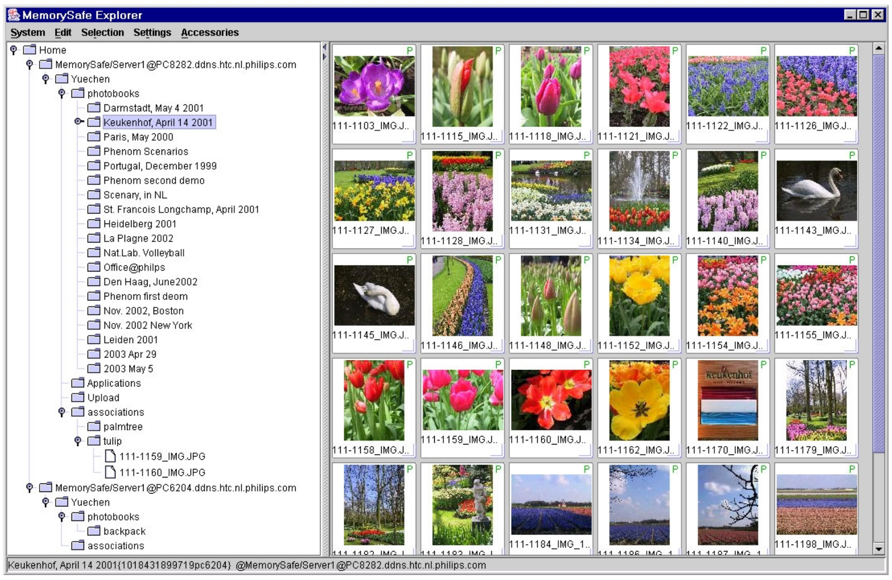


Figure 8.1. A snapshot of the MemorySafe Explorer.

By default, a user of a MemorySafe server sees only his or her own root directory when using the MemorySafe Explorer. The reason for this is that managing digital photos is considered to be a personal activity. The user can choose to see all the users of a server. The user can thus browse all public data of the other users of the server. Typically, a user will selectively make a few photo albums public so that all the users of the system can browse them. When using the MemorySafe Explorer, the user can modify only his or her own data. He or she may browse public data of another user and is not allowed to modify them. The user can make a copy of the public data of another user via drag-and-drop operations.

In the server-transparent access mode, the MemorySafe Explorer permits a user to browse and manipulate data, regardless of the locations where his or her data are stored. In this mode, the MemorySafe Explorer will collect personal data from the MemorySafe servers available in the system and present them to the user as if the user is working with a single MemorySafe server. In this way, the user can easily examine personal data. The user can also manage his or her data when using the MemorySafe Explorer in the server-transparent mode. In this mode modifications on the data will first be applied to the user's "home" server and then to the "local" server, if such a server is available. The user can always switch the working mode of the MemorySafe Explorer to the server-aware access mode.

Semantics of drag-n-drop operations. By using the MemorySafe Explorer, a user can manipulate his or her data via drag-n-drop operations.

Drag-n-drop operations have different semantics. Four semantics have been identified. They are:

- *Linking.*The user creates a new link between a file and a directory. For example, the user wants to add a photo to a newly created album. He or she does so by dragging the icon of the photo and dropping it onto the icon of the album.
- *Copying.* The user creates a copy of a resource. The copy has the same data as the original resource, but it has a different resource identity. For example, the user wants to make a new copy of a photo.
- *Replicating.* The user replicates a resource from a MemorySafe server to another one. For example, the user wants some of his or her photo collections to be available on a portable device. To this end, he or she drags a collection of photos from a MemorySafe server representing a server at home and drops it onto the MemorySafe server representing the portable device.
- *Moving.* The user wants to move a resource from one directory to another, possibly from one MemorySafe server to another server. For example, the user moves all data stored on a portable device to a server at home.

When using the MemorySafe Explorer, a user can choose the semantics of a drag-n-drop operation. The actual result of a drag-n-drop operation will be determined by the context in which the operation is performed. The context of a drag-n-drop operation consists of the following elements: the owner of the resource being dragged and the server on which the resource is stored, the owner of the resource on which a resource is dropped and the server of the resource on which a resource is dropped.

Table 8.1 shows the semantics of a drag-n-drop operation in different contexts. Suppose a resource A is dragged and dropped onto a resource B, where B is a directory resource.

- The linking semantics is only applicable when A and B are stored on the same server and belong to the same user.
- The copying semantics can always be used.
- The replicating semantics can only be used when A and B belong to the same user but are stored in different MemorySafe servers.
- The moving semantics can only be used when A and B belong to the same user.

A synthesized semantics is defined and set to be the default semantics of a drag-n-drop operation on the MemorySafe Explorer, as shown in Table 8.1. This semantics aims to conform to the default semantics of a drag-n-drop operation on a Windows system, in order to allow people who are familiar with Windows systems to quickly understand the result of the operation without learning.

Still, the default semantics of a drag-n-drop operation on the MemorySafe Explorer is different from that on a Microsoft Windows system.

Table 8.1. The result of a drag-n-drop operation on the user interface of the MemorySafe Explorer. Suppose a resource A is dragged and dropped onto a resource B. The result of this drag-n-drop operation will depend on the selected semantics of the drag-n-drop operation, whether A and B belong to the same user and whether A and B are stored on the same MemorySafe server.

Semantics	The same owner		Two different owners	
of drag-n-drop	Single server	Two servers	Single server	Two servers
Linking	Linking	-[a]	-	-
Copying	Copying	Copying	Copying	Copying
Replicating	-	Replicating	-	-
Moving	Moving	Moving	-	-
Default	Moving	Replicating	Copying	Copying

[a] Not allowed.

Table 8.2. Comparison of a drag-n-drop operation on the user interface of the MemorySafe Explorer and a drag-n-drop operation on the user interface of a Microsoft Windows system. Suppose a resource A is dragged and dropped onto a resource B. The result of this drag-n-drop operation will depend on the selected semantics of the drag-n-drop operation, whether A and B belong to the same user and whether A and B are stored on the same devices.

Semantics	The same owner		Two different owners	
of drag-n-drop	Single server	Two servers	Single server	Two servers
Default MSW[a]	Moving	Copying	Moving	Copying
Default MSE[b]	Moving	Replicating	Copying	Copying

[a] On the user interface of a Microsoft Windows system.
[b] On the user interface of the MemorySafe Explorer.

- On a Windows system, dragging a file and dropping it onto another device has the copying semantics while on the MemorySafe Explorer the same operation will have the replicating semantics. In the MemorySafe system, the identity of the resource that was dragged will be preserved after the drag-n-drop operation. This means that the user will later easily be able to check whether the resources from two servers have the same origins.
- On a Windows system, dragging a file and dropping it onto another user's directory on the same device has the moving semantics and doing so between devices have the copying semantics. On the MemorySafe Explorer dragging a file and dropping it onto another user's directory always has the copying semantics. The reason for this is that in a MemorySafe server, different users have different root directories and a user is not allowed to modify another user's data.

The comparison of the default drag-n-drop semantics of a drag-n-drop operation on the MemorySafe Explorer and on a Microsoft Windows system is summarized in Table 8.2.

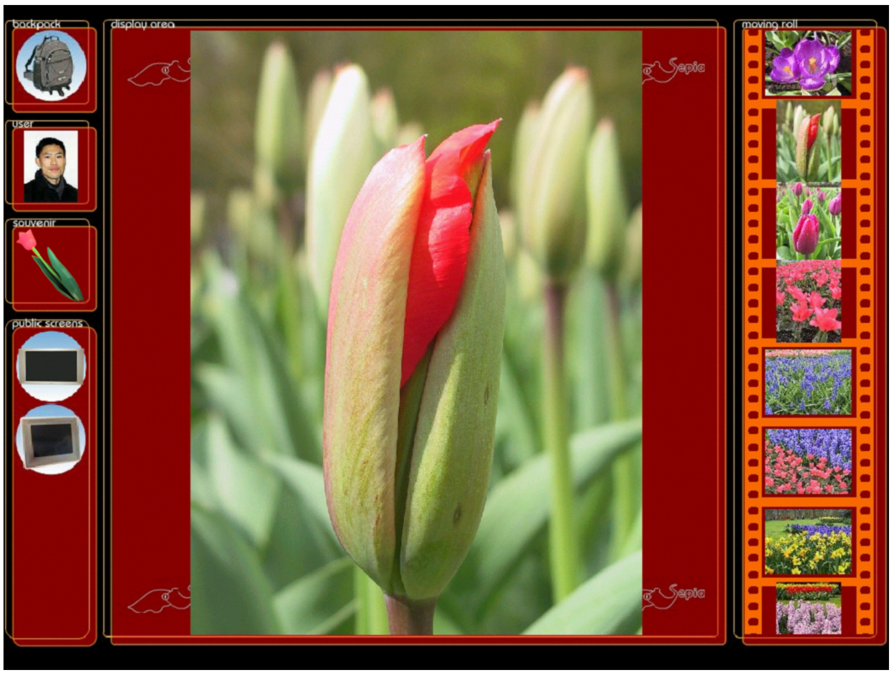


Figure 8.2. A screen shot of the user interface of the Sepia in the "Memory Sharing" demonstration system.

8.2 Memory Sharing

The "Memory Sharing" system was developed by the Phenom team to demonstrate how people could exchange digital photos in future intelligent home environments. The MemorySafe system is responsible for data management in this demonstration system.

Figure 8.2 illustrates the user interface (UI) of the portable device (Sepia) with which a user can browse personal photo albums and exchange photos with others. Sepia has a touch screen, with which a user can perform the following actions.

A1. Drag the thumbnail of a digital photo from the "film roll", located on the right side of the screen, and drop it on the central part of the screen to view the photo.

A2. Drag the thumbnail of a digital photo from the "film roll" (or the photo at the center) and drop it on any screen icon on the left side of the screen, to display the photo on the target screen.

A3. Drag the thumbnail of a digital photo from the "film roll" (or the photo in the center) and drop it on the "backpack" icon, located in the top left corner of the screen. The photos put in the "backpack" will be stored in the local storage of the Sepia and become available even when the

Sepia is disconnected from the home. In this way, the user will be able to continue to access this photo on the move.

A4. Drag the thumbnail of a digital photo from the "film roll" (or the photo at the center) and drop it on the object icon in the middle of the left side of the screen. The object icon stands for a souvenir object available on the "chameleon" table, a table capable of detecting objects with radio frequency tokens. The drag-n-drop operation creates an association between the photo and the object.

The above-mentioned interactions were implemented in Java, with the help of the "UIManager", a generic component of the Empire system for creating a range of user interfaces supporting drag-n-drop operations. MemorySafe provides the following supports. Note that each item in the following list corresponds to the item with the same number in the action list just described.

A1. When the thumbnail of a photograph is dragged from the "film roll" and dropped at the center of the screen, the GURI of the digital photo will be sent to the MemorySafe system. If the GURI exists, a resource handler of the requested resource will be sent back to the Sepia UI. With this resource handler, the Sepia UI will fetch the image data from the MemorySafe system. In the "Memory sharing" demonstration, a Sepia is a personalized device and the "film roll" shows only personal digital photos of the person currently using the Sepia. This drag-and-drop operation hence has the access right to the MemorySafe system. If the resource does not exist, an error message will be displayed on the screen of the Sepia.

A2. When the thumbnail of a digital photo is dragged and dropped onto a screen icon, the GURI of the thumbnail, together with the access right of the current user of the Sepia, will be sent to the displaying device. Using the received GURI and access right, the displaying device will fetch data from the MemorySafe system.
Note that devices in the Empire system are divided into different administrative groups. For example, all devices including displaying devices at Mark's home belongs to one group, say "Mark", as illustrated in Figure 8.3. When Clair is visiting Mark, Clair's Sepia, belonging to the group "Clair", will appear as a screen icon on Mark's Sepia. When Mark drags and drops a picture on the screen icon representing Clair's Sepia, Clair's Sepia will receive the GURI of the photo. However, Clair's Sepia won't resolve this GURI on its local MemorySafe server, because Clair's Sepia does not belong to Mark's device group. Therefore, Clair's Sepia will forward the GURI resolving request to all neighboring device groups. When one photo resource with the GURI has been found, the photo will be displayed on the screen of Clair's Sepia.

A3. When the thumbnail of a digital photo is dragged and dropped onto the "backpack" icon, the photo will be copied to the local MemorySafe server in an identity-preserving manner if the dragged image belongs to

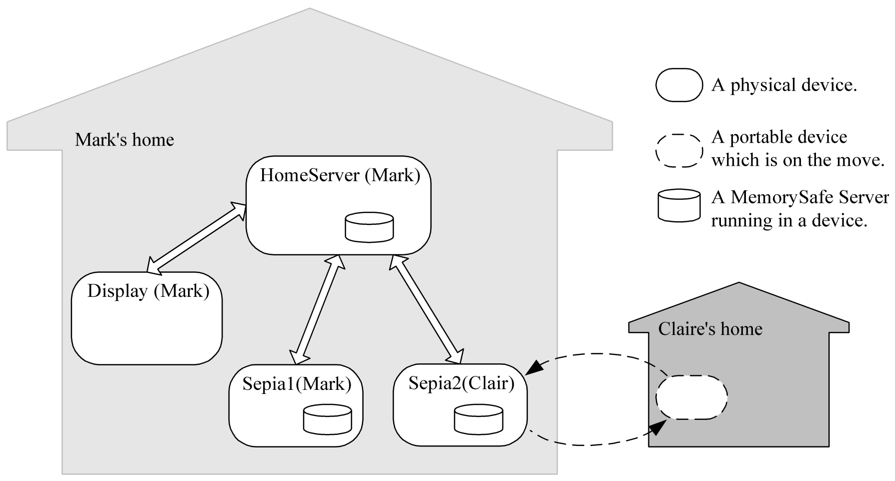


Figure 8.3. Devices belonging to different groups communicate with each other.

the current user of the Sepia. If the dragged photo is one which was received from another user, a new resource will be created on the local MemorySafe server for storing the received photos.

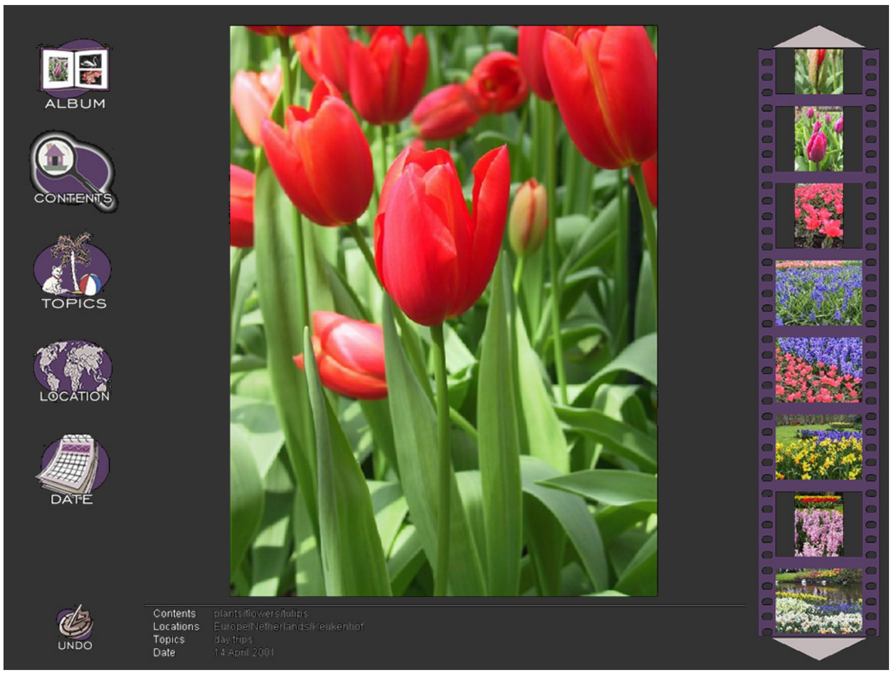
A4. When the thumbnail of a digital photo is dragged and dropped onto a souvenir object, Sepia UI will search for a directory storing all associations with the object in the MemorySafe system. If the directory for storing the associations does not exist, the directory resource will be created. If the directory is found or created, a link to the dragged photo will be created in the directory resource. If the dragged photo is from another user, a new resource with the identical content of the dragged photo will be created on the MemorySafe server.

In programming the Memory Sharing system, data are accessed in a server-transparent manner. Sepia UI simply sends server-transparent requests such as:

- `getResource(String guri)`,
- `getResourceByPath(String path)`,
- `drag_n_drop(String src_guri, String tgt_guri)`.

The semantics of such operations are easily explained to the developers and behaviors of applications developed using such methods are easily understood by end users.

Memory Sharing was successfully demonstrated as a part of the “HomeLab Tour” of the Philips Corporate Research Exhibition 2003. Throughout this event, the MemorySafe system worked reliably and no problems were encountered.



Contents	plants/flowers/tulips
Locations	Europe/Netherlands/Keukenhof
Topics	day trips
Date	14 April 2001

Figure 8.4. A screen shot of the user interface of the Browsing Assistant.

8.3 Browsing Assistant

Browsing Assistant [45] is a photo browsing tool developed by J.M.A. Ferreira, which allows users to wander through personal photo collections in several distinct dimensions. Users can look at photographs with the same contents, photographs taken at the same location or photographs of the same category. Figure 8.4 illustrates the user interface of Browsing Assistant.

Browsing Assistant was developed on top of the MemorySafe system. The MemorySafe system stores and provides information that is used by Browsing Assistant. In the MemorySafe system, digital photos are stored in the system as resources. Users can provide metadata (semantic information on photos) by entering the information in the system using the MemorySafe Explorer. Metadata are treated as part of the resource description of a resource storing a digital photo.

Table 8.3 lists all the descriptors that are used in the MemorySafe system to describe different aspects of a photo. That list of descriptors does not cover all the aspects of a digital photo. Instead, it contains only the aspects that are of research interest to conversational search [154]. In practice, the MemorySafe system does not impose any restrictions on the descriptors defined by users. The users can add anything to the system, according to their own needs.

Table 8.3. The keys of descriptors of a resource that store a digital photo in the MemorySafe system.

Key of descriptor	Purposes
`Contents`	Describes entities, such as persons, buildings and objects, which can be visually identified in the photograph.
`locations`	Describes the place where the photograph was taken.
`topics`	Describes classifications of the photograph.
`date`	Describes the date when the photograph was taken.
`name`	Describes the textual name for recalling.
`title`	Describe the title of the photograph, which is often indicated on the user interface.
`description`	Describes anything that the user might put into the system, for example annotations.

When collecting metadata of digital photos in the MemorySafe system, Browsing Assistant obtains resource handlers in return. Each resource handler caches the resource description and the data of the thumbnail, which helps to reduce network traffic and improve GUI rendering of Browsing Assistant. In maintaining cache consistency, Browsing Assistant performs periodic checks to ensure the validity of cached data.

8.4 Photo concept browsing

Photo Concept Browsing [123] is another photo-browsing tool built on top of the MemorySafe system, which enhances user experiences of browsing digital photographs. This tool integrates *Formal Concept Analysis* [47], a theory focusing on the inner conceptual structure of data and providing graphic presentations of concept structures, in photo browsing. Details of the design and implementation of the tool will be presented in Chapter 9.

8.5 Managing application-specific data

The MemorySafe system is also used for managing application-specific data, besides digital photos. For example, the user interface of the Phenom "Memory Sharing" demonstration system and the Browsing Assistant are created by using the UIManager. Definitions and configurations of the user interfaces are stored in the MemorySafe system.

The MemorySafe system records modifications to application-specific data and application developers can always retrieve old versions of data. During application development, an application may write incorrect data into its configuration. Using the MemorySafe Explorer, developers can retrieve a proper version of the setting which is stored in the access history of the resource of the configuration.

8.6 User experiences

The MemorySafe system was deployed in the Phenom development environment in July 2002. Since then, the system has been used by the members of the Phenom development team. Feedback from the users of the MemorySafe system has been positive, with the emphasis on the following aspects of the MemorySafe system.

- In the MemorySafe system, introducing data identities and aggregating various aspects and copies of a data object into a resource simplifies logical views and proves to be very useful. This feature is highly appreciated by users.
- The linking mechanism provided by the MemorySafe system nicely fulfills the requirement of associating digital photos with souvenir objects. Moreover, it also allows links to a directory and prohibits multiple links to a file in the same directory. In this way it makes the link mechanism more useful.
- Server-transparency provides a simple view of personal data stored in different devices. The MemorySafe system provides a server-transparent access interface for application development, which facilitates programming.

The MemorySafe system had some performance problems in its initial implementation. Descriptions of resources were stored in separate files. When the Phenom's Browsing Assistant collected metadata of digital photos from the MemorySafe system, loading resource descriptions from the files stored on hard disks prolonged the waiting time for the users of Browsing Assistant. This problem was later solved by indexing and caching resource descriptions.

Some aspects of the MemorySafe system were not fully explored. For example, server grouping is a featuring mechanism of the MemorySafe system for managing device on/off and entering/leaving activities. This mechanism had not yet been fully tested and its design had not been validated. This was due to the rather static configurations of the Phenom demonstration systems.

9. Photo concept browsing

PHOCON[1] (Photo Concept Browsing) is an innovative photo browsing and searching tool built on top of the MemorySafe system. This tool uses *Formal Concept Analysis* [47] to analyze correlations between photos with metadata and provides a user interface enabling people to easily navigate photos in an automatically generated hierarchy. In this chapter the design and implementation of PHOCON are presented.

9.1 Motivation

With the advent of digital photography and the availability of sufficiently large storage devices for home applications, users will be faced with large collections of personal digital photos. There will be a need for new types of user-interfaces for browsing photo collections [1, 72, 9, 129, 12, 101].

In the MemorySafe system, users can browse personal photos per album and retrieve photos by using souvenir objects. Likewise, using software such as ACDSee [2], people can browse photos stored in directories of local file systems in a per-directory manner. With the help of album-composing tools such as PhotoParade, FipAlbum and 3D-Album Photo Organizer, people will be able to view photos in predefined photo albums, a special type of presentation files containing photos or references to photos. In all these cases, users will have limited options to browse personal photos and there will be little support for cross-directory or cross-album browsing, which will make photo browsing activities less attractive and less fun.

In the MemorySafe system, users can annotate and categorize photos and can search for photos on the basis of certain criteria, such as keyword matching. In a similar way, people can annotate and categorize their photos stored in different directories, with software support. People can search for photos using keywords. In both cases, when searching for photos, a user will often receive a lengthy list of photos matching the specified keywords. The matching

[1] This tool is an improved version of ICE (Interactive Concept Lattice Exploration) [123]. As was found in the author's further study of Formal Concept Analysis, the term "exploration" is used as a mechanism to build up concept lattices in formal concept analysis. To avoid confusion, the term "navigation" was adopted for the improved version of ICE.

results are usually presented in a lexical or statistical order, which does not meet the user's expectations with respect to meaningful ordering or grouping. Moreover, given a large collection of photos, keyword-based searching often proves to be a long process requiring iterative searching steps, which may exhaust the user's and the spectator's interest in browsing photos in home environments.

Formal Concept Analysis is based on a binary relation between *objects* and *attributes*. It provides methods for grouping objects and attributes into *concepts*, pairs of object sets and attribute sets, so that the binary relation can be presented in a hierarchical mathematical structure, a so-called *concept lattice* [173]. Such lattices can be pictorially represented in line diagrams. Formal Concept Analysis is a mathematical tool used in data analysis and knowledge engineering. It has also been applied in information systems to improve data presentation.

Formal Concept Analysis can be used to improve user experience of browsing and searching digital photos. Through categorization and annotation, conceptual knowledge related to digital photos is transferred to the system and is explicitly associated with the image files of the photos. Usually such associations will be made between image files and keywords. The relation between image files and keywords can be modelled as a context, with objects being digital photos and attributes being keywords. A concept lattice can be automatically derived from a context. Digital photos can consequently be viewed in a concept-wise manner, in addition to per-directory browsing.

As far as searching for photos is concerned, search criteria can be formulated as a list of keywords with which a photo concept can be computed. This photo concept not only contains all the photos with the specified keywords, but also reveals the other keywords shared by the photos. In this way, suggestive information is provided to users for further searching and browsing. Moreover, closely related concepts in the concept lattice can be presented to users, allowing them to relax or intensify search criteria to locate desired photos simply by browsing the concept lattice.

9.2 Formal Concept Analysis

Formal Concept Analysis starts with a *context*. A context $\mathbb{K}$ is a triple (G, M, I), where G and M are two sets and $I \subseteq G \times M$ is a relation between G and M. The elements of G are called the *objects* of the context. The elements of M are called the *attributes* of the context. An element (g, m) of I indicates that the object g has the attribute m.

Table 9.1 illustrates an example context. The relation between image files and categories is modelled as a context, with objects being image files and attributes being category names.

Given a set $A \subseteq G$ of objects, A' denotes all the attributes that are common to the objects in A.

Table 9.1. Context of a collection of photos. Photos are regarded as objects and their user-defined categories are regarded as attributes.

		a	b	c	d	e
		Family	Home	Pet	Vacation	Summer
1	DSCN0010	x	x			
2	DSCN0019	x	x			
3	DSCN0101	x	x	x		
4	DSCN0152	x		x		
5	DSCN0210	x	x		x	
6	DSCN0340	x	x	x	x	
7	DSCN0456	x		x	x	x
8	DSCN1024	x		x	x	

$$A' = \{m \in M \mid \forall\ g \in A : (g, m) \in I\}$$

Given a set $B \subseteq M$ of attributes, correspondingly, B' denotes all the objects that are common to the attributes in B.

$$B' = \{g \in G \mid \forall\ m \in B : (g, m) \in I\}$$

Note that when A is an empty set, $A' = M$. Likewise, when B is an empty set, $B' = G$. Take the context in Table 9.1 as an example.

$$\{\text{DSCN0101}\}' = \{\text{Family}, \text{Home}, \text{Pet}\}$$
$$\{\text{Family}, \text{Home}, \text{Pet}\}' = \{\text{DSCN0101}, \text{DSCN0340}\}.$$

For simplicity, abbreviations of object and attribute names are used. The above equations can hence be written as

$$\{3\}' = \{\text{a}, \text{b}, \text{c}\},$$
$$\{\text{a}, \text{b}, \text{c}\}' = \{3, 6\}.$$

A *concept* of a context (G, M, I) is a pair (A, B), where $A \subseteq G$, $B \subseteq M$, $A' = B$ and $B' = A$. A and B are called the *extent* and the *intent* of the concept (A, B), respectively. For example, $(\{3, 6\}, \{a, b, c\})$ is a concept. Table 9.2 lists all the concepts of the context in Table 9.1.

Formally, it can be proven that given any subset $A \subseteq G$, (A'', A') is a concept; correspondingly, given any subset $B \subseteq M$, (B', B'') is also a concept. For example, $(\{3\}'', \{3\}')$ is a concept. It can be verified that $(\{3\}'', \{3\}') = (\{3, 6\}, \{a, b, c\})$. For an object g, $(\{g\}'', \{g\}')$ is called the *object concept* of g. Likewise, for an attribute m, $(\{m\}', \{m\}'')$ is called the *attribute concept* of m.

Given two concepts, (A_1, B_1) and (A_2, B_2), of a context, a relation $\leq$, called *hierarchical order*, is defined as follows: $(A_1, B_1) \leq (A_2, B_2) \Leftrightarrow A_1 \subseteq A_2$. Equivalently, $(A_1, B_1) \leq (A_2, B_2) \Leftrightarrow B_1 \supseteq B_2$ holds. (A_1, B_1) is called a *subconcept* of (A_2, B_2) and (A_2, B_2) is called a *superconcept* of (A_1, B_1).

Table 9.2. Concepts of the context in Table 9.1.

concepts	objects	attributes
i	1,2,3,4,5,6,7,8	a
ii	1,2,3,5,6	a,b
iii	3,4,6,7,8	a,c
iv	5,6,7,8	a,d
v	3,6	a,b,c
vi	5,6	a,b,d
vii	6,7,8	a,c,d
viii	6	a,b,c,d
ix	7	a,c,d,e
x		a,b,c,d,e

Moreover, $(A_1, B_1) < (A_2, B_2)$, if $(A_1, B_1) \leq (A_2, B_2)$ and $A_1 \neq A_2$. The set of all the concepts of (G, M, I), denoted by $\underline{\mathfrak{B}}(G, M, I)$, is hierarchically ordered by $\leq$ and is called the *concept lattice* of the context (G, M, I). Concept lattices prove to be complete lattices: given any two concepts, there always exists a common superconcept and a common subconcept; for any set of concepts, there always exists a largest element in the set of all common subconcepts and a smallest element in the set of all common superconcepts. The line diagram depicted in Figure 9.1 illustrates the concept lattice of the context defined in Table 9.1.

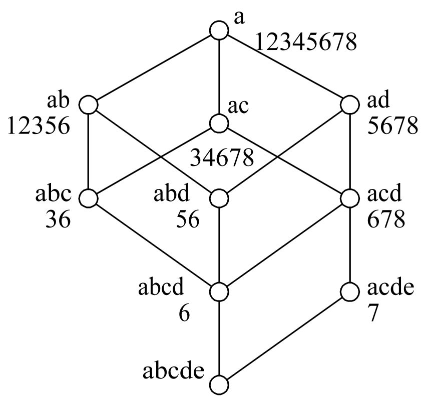


Figure 9.1. The concept lattice of the context defined in Table 9.1.

9.3 Design of PHOCON

PHOCON is designed to enable normal users to browse and search their photos easily. Ease-of-use is an important requirement in its design. Both the interface of PHOCON and the interaction between PHOCON and a user should be simple and intuitive. Also, its use should not require scientific knowledge or professional expertise.

9.3.1 Line diagrams

Line diagrams and their variants have been extensively used in visualizing concept lattices. Depending on the contexts in question, however, line diagrams may become too complicated to read and, consequently, may become less helpful for users in their attempts to understand concept lattices. This will be the case especially when line diagrams are used to present browsing or search results to end users, without any rendering.

In an earlier experiment, photos that had been taken during a social event, referred to as the "3rd EESI Happening", were used as the sample data. In total, 88 photos were taken. Each photo was annotated by identifying the persons in the photos, the location where the photo was taken and the activity at the moment of shooting. In total, 43 keywords were used. On average, there were 7.761 (≈ 8) attributes per object.

The ConExp tool [175] was used in data analysis. The sample photos were treated as objects and the specified metadata (keywords) were regarded as attributes. With the help of ConExp, 298 concepts were discovered and a line diagram of the concept lattice was drawn. The line diagram is given in Figure 9.2. As can be seen in the figure, the line diagram becomes almost unreadable in the case of a large context. Similar experiences are reported for concept analysis of legacy systems [82].

9.3.2 Direct subconcepts and direct superconcepts

When viewing photos, people often want to browse and look at other photos in an associative manner: they are also interested in photos that are somehow related to the ones currently being viewed, for example according to content.

The photos currently being viewed can be treated as a concept. So the photos that are in one way or another related according to their metadata can be grouped into concepts and presented in a concept lattice. In most cases, only the superconcepts and subconcepts of the "current" concept will be of relevance in photo browsing, with the direct superconcepts and subconcepts forming the closest neighbors of the concept and providing links to the other concepts. It will suffice here to present only the direct superconcepts and subconcepts of a concept. In this way, a full line diagram can be reduced to a simple representation which appears to be convenient and intuitive for end users and can be easily implemented.

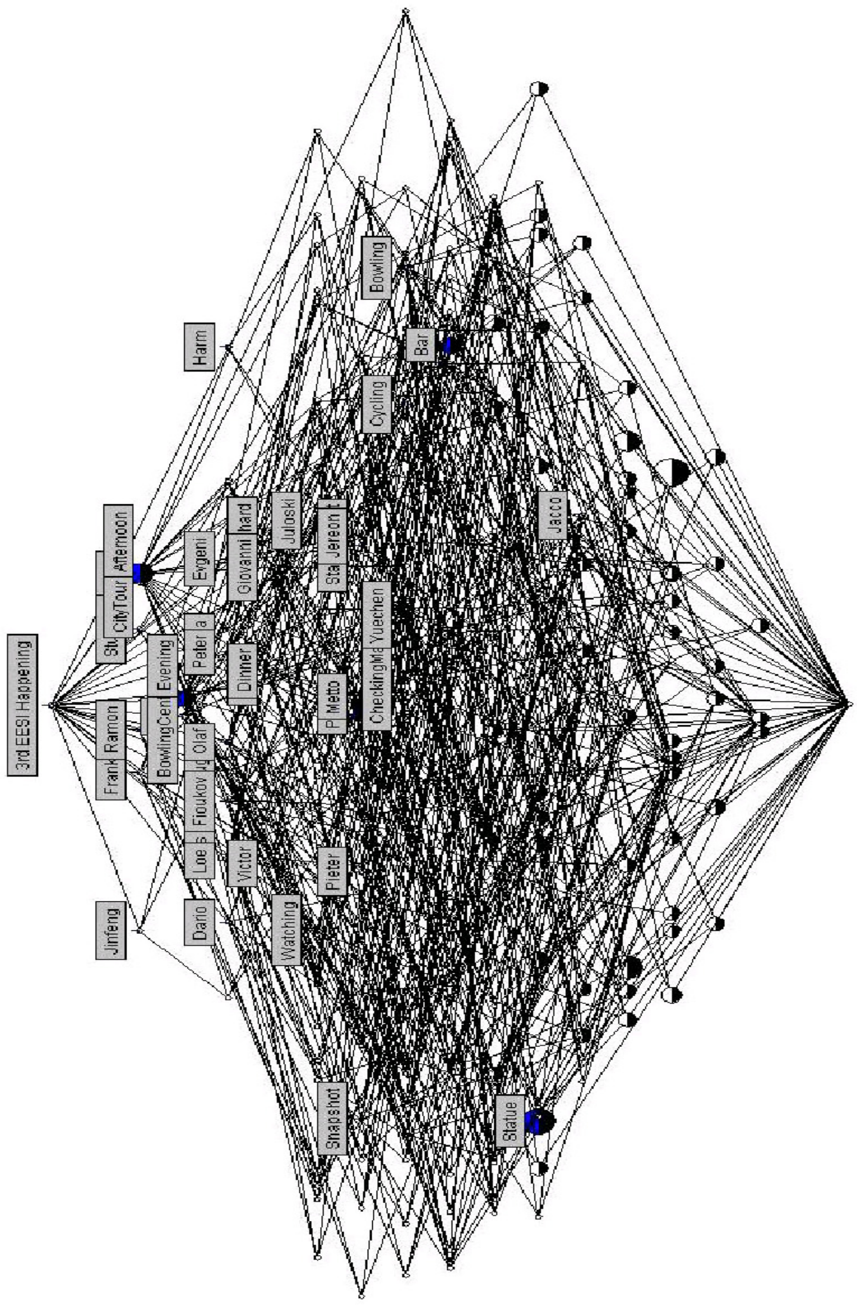


Figure 9.2. The concept lattice of the context that describes a collection of photos with medadata. The photos were taken during a social event referred to as “3rd EESI Happening” and were manually annotated. The line diagram of the concept lattice was generated by ConExp.

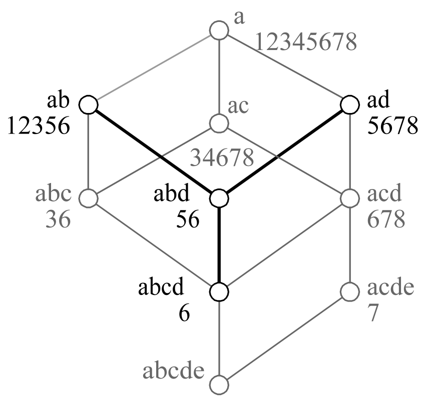


Figure 9.3. The direct superconcepts and subconcepts of the concept $(\{5,6\},\{\mathrm{a},\mathrm{b},\mathrm{d}\})$ in the context defined in Table.9.1.

Given two concepts, (A_1,B_1) and (A_2,B_2), of a context (G,M,I), (A_1,B_1) is called a *direct subconcept* of (A_2,B_2), if (A_1,B_1) is a subconcept of (A_2,B_2) and there is no other concept (A_3,B_3) such that $(A_1,B_1) < (A_3,B_3)$ and $(A_3,B_3) < (A_2,B_2)$. In this case, (A_2,B_2) is a *direct superconcept* of (A_1,B_1), written as $(A_1,B_1) \prec (A_2,B_2)$. The relation $\prec$ can be formally defined as follows[2].

$$\begin{aligned}(A_1,B_1) \prec (A_2,B_2) \Leftrightarrow (&A_1 \subseteq A_2 \wedge \neg\exists(A_3,B_3) \in \underline{\mathfrak{B}}(G,M,I) : \\ &A_3 \neq A_1 \wedge A_3 \neq A_2 \wedge A_1 \subseteq A_3 \wedge A_3 \subseteq A_2)\end{aligned}$$

The direct subconcept and superconcept relation is used to draw line diagrams to visualize concept lattices. In a line diagram, concepts are depicted by circles. Given any two concepts, (A_1,B_1) and (A_2,B_2) of the concept lattice of a context (G,M,I), where $(A_1,B_1) \prec (A_2,B_2)$, the circle representing (A_2,B_2) is depicted above the circle representing (A_1,B_1), and the two circles are joined by a line segment. Figure 9.3 shows the direct superconcepts and subconcepts of the concept $(\{5,6\},\{\mathrm{a},\mathrm{b},\mathrm{d}\})$ in the context defined in Table 9.1. Only the direct superconcepts and subconcepts of a given concept are displayed to achieve simplicity in lattice presentation.

9.3.3 Algorithms

Given a concept (A,B) of a context (G,M,I) with $A \subseteq G$ and $B \subseteq M$, all subconcepts, other than (A,B) itself, can be obtained in the following way:

$$(C', C'') \text{ for every } C, \text{ where } B \subset C \subseteq M.$$

[2] In the literature [92], *lower neighbors* and *upper neighbors* have been used for naming direct subconcepts and direct superconcepts, respectively.

Given any two supersets C and D of B, where $C \subset D$, it can be proven that $(D', D'') \leq (C', C'')$. Therefore, (D', D'') is not a direct subconcept of (A, B), if $D' \neq C'$. So in practice, the previous method is refined as follows.

$((B \cup \{m\})', (B \cup \{m\})'')$ for every $m \in M - B$.

The actual algorithm is as follows[3].

Algorithm for computing direct subconcepts. Declare a variable set *Subs* for storing all the direct subconcepts. Initially, $Subs = \emptyset$. Next, for each attribute $m \in M - B$, let

$Z = (B \cup \{m\})'$.

If $Subs = \emptyset$, simply add (Z, Z') to *Subs*; otherwise, proceed to check the following:

1. If there exists a concept (X, Y) in *Subs* such that $X \subset Z$, remove (X, Y) from *Subs* and add (Z, Z') to *Subs*.
2. If there exists a concept (X, Y) in *Subs* such that $Z \subseteq X$, do nothing.
3. In all other cases, add (Z, Z') to *Subs*.

Note that $X \subset Z$ if $X \subseteq Z$ and $X \neq Z$. Moreover, (Z, Z') is a concept.

To calculate the direct subconcepts of $(\{1,2,3,4,5,6,7,8,\}, \{\mathrm{a}\})$, the algorithm works as follows. First, let $m = \mathrm{b}$. Since $Subs = \emptyset$, $(\{1,2,3,5,6\}, \{\mathrm{a}, \mathrm{b}\})$ is added to *Subs*. Next, let $m = \mathrm{c}$. $(\{3,4,6,7,8\}, \{\mathrm{a}, \mathrm{c}\})$ is added to *Subs*, since it is neither a subconcept nor a subconcept of any concept in *Subs*. For the same reason, $(\{5,6,7,8\}, \{\mathrm{a}, \mathrm{d}\})$ is added to *Subs*. Finally, let $m = \mathrm{e}$. In this case, $Z = \{7\}$. As illustrated in Figure 9.4, Z is a subset of the extent of $(\{3,4,6,7,8\}, \{\mathrm{a}, \mathrm{c}\})$ and $(\{5,6,7,8\}, \{\mathrm{a}, \mathrm{d}\})$. So e is ignored. Finally, *Subs* contains all the direct subconcepts of $(\{1,2,3,4,5,6,7,8,\}, \{\mathrm{a}\})$.

Similarly, an algorithm for calculating all the direct superconcepts of (A, B) has been developed.

Algorithm for computing direct superconcepts. Declare a variable set *Sups* for storing all the direct superconcepts. Initially, $Sups = \emptyset$. Next, for each object $g \in G - A$, let

$Z = (A \cup \{g\})'$.

If $Sups = \emptyset$, simply add (Z', Z) to *Sups*; otherwise, proceed to check the following:

1. If there exists a concept (X, Y) in *Sups* such that $Y \subset Z$, remove (X, Y) from *Sups* and add (Z', Z) to *Sups*.
2. If there exists a concept (X, Y) in *Sups* such that $Z \subseteq Y$, do nothing.
3. In all other cases, add (Z', Z) to *Sups*.

Note that (Z', Z) is a concept.

[3] Neighbor algorithms can be used for computing direct superconcepts and direct subconcepts [92].

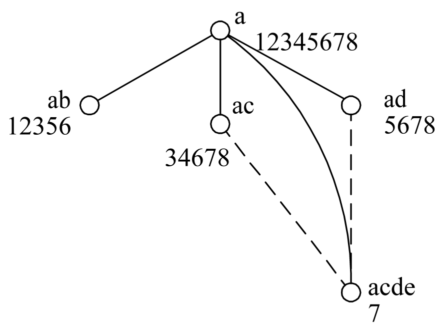


Figure 9.4. An intermediate step in calculating direct subconcepts of the concept $(\{1, 2, 3, 4, 5, 6, 7, 8\}, \{a\})$.

9.3.4 Step-wise concept lattice navigation

On the basis of the direct subconcept and superconcept relation, concept lattices can be browsed in a step-wise manner. Figure 9.5 illustrates a concept lattice navigation from the top concept to the concept $(\{6\}, \{\mathrm{a, b, c, d}\})$, following the path $a \to d \to c \to b$. In each step, both the superconcepts and the subconcepts of the concept being visited are generated for further navigation.

In step-wise concept lattice browsing users may follow different paths to locate a concept. As can be seen in Figure 9.1, the concept $(\{6\}, \{\mathrm{a, b, c, d}\})$ can be reached by other paths, such as the path $a \to b \to c \to d$. This convenient property makes for flexibility for users in browsing concept lattices of photos.

Moreover, the fact that there is no need to calculate a full lattice beforehand saves both user time and computing memory. In home environments, the number of digital photos of a user may increase dramatically. So building a full concept lattice of the photos may become a time- and resource-consuming process, as indicated in [86]. This problem is avoided in step-wise lattice navigation.

9.3.5 Searching

Step-wise concept lattice navigation can be combined with traditional search methods. For example, in Figure 9.5 a user might look for photos associated with the keyword “summer” (e) right after the “Expand a” step. Examining the three direct subconcepts, the user might think that there are no photos associated with the keyword “summer” at all. This is certainly not the case. As can be seen in Figure 9.4, the concept $(\{7\}, \{\mathrm{a, c, d, e}\})$ is discarded during the generation of the direct subconcepts of $(\{1, 2, 3, 4, 5, 6, 7, 8, \}, \{\mathrm{a}\})$, to achieve compactness.

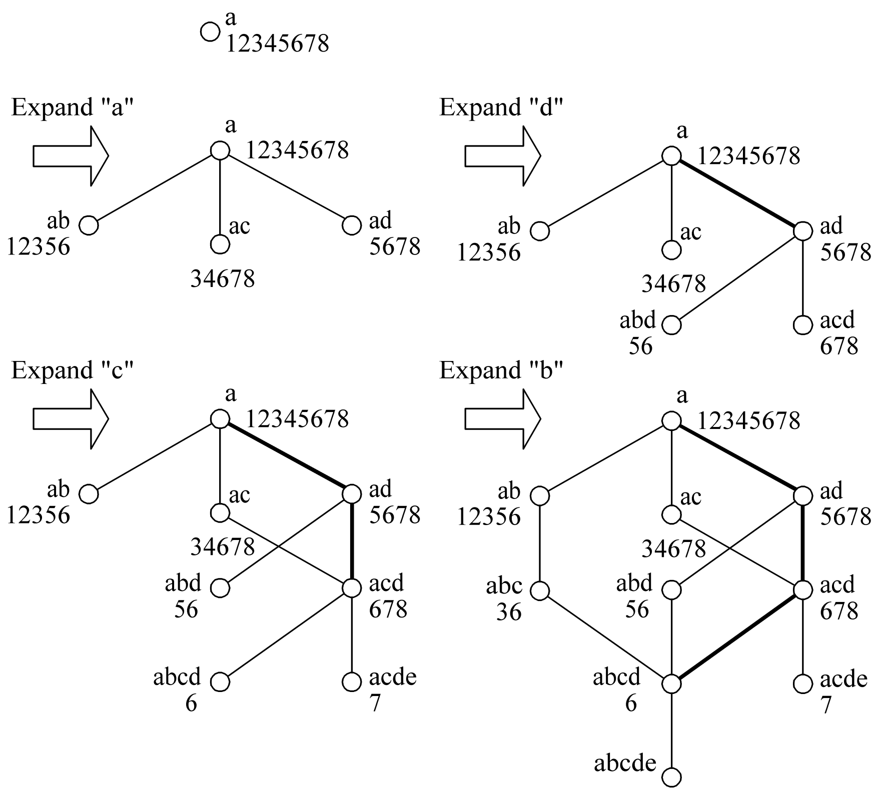


Figure 9.5. Step-by-step navigation of the concept lattice in Figure 9.1. The navigation follows the path $a \rightarrow d \rightarrow c \rightarrow b$.

One solution to allow the user to locate $(\{7\}, \{\mathrm{a, c, d, e}\})$ quickly would be to lift $(\{7\}, \{\mathrm{a, c, d, e}\})$ so that it could be reached directly via the top concept $(\{1, 2, 3, 4, 5, 6, 7, 8, \}, \{\mathrm{a}\})$. However, this treatment would make the interface inconsistent and would introduce redundancy. An alternative is the integration of searching. In the above example, the photo DSCN0456 (7) can be found directly by searching for the keyword “summer”. In this way no redundancy is introduced. This second approach was consequently implemented in PHOCON.

In presenting the search result, the concept $(\{7\}, \{\mathrm{a, c, d, e}\})$ can be shown to the user to indicate all the other associated keywords. In general, given a set B of user-specified keywords,

$$((B \cap M)', (B \cap M)'')$$

will be used as the resulting concept for presentation. Moreover, its subconcepts and superconcepts will also be displayed to enable strengthening and weakening of search criteria, respectively. The subconcepts and superconcepts can be obtained using the algorithms described in Section 9.3.3.

9.4 Implementation of PHOCON

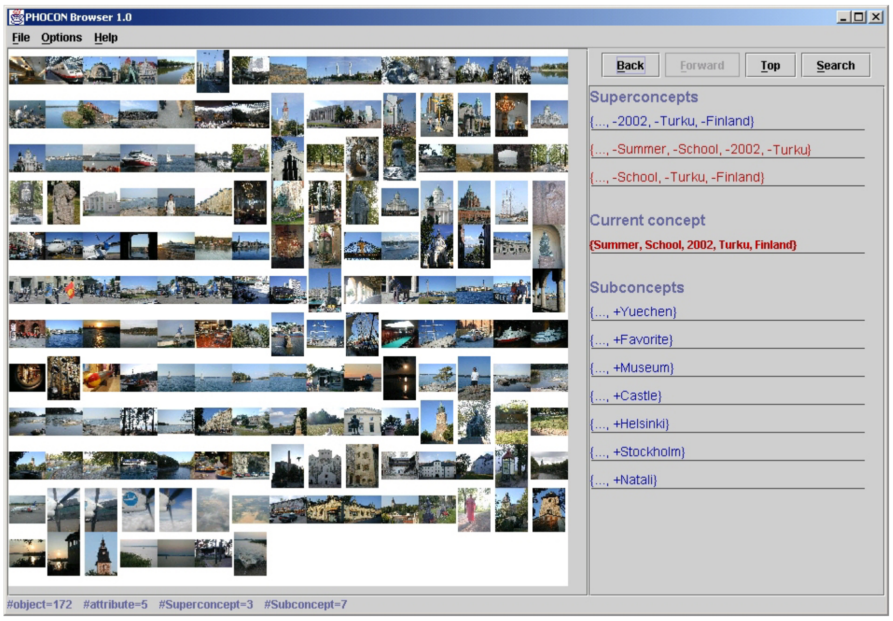
The MemorySafe system was developed to enable end users to manage digital photos in home environments [121]. PHOCON was implemented in Java and integrated as a tool of the MemorySafe system. The algorithms that were described in Section 9.3.3 were coded using Java (JDK1.2). The main data structures, *Subs* and *Sups*, were represented by dynamic arrays (Vectors) in Java.

All the photos and their user-defined keywords in the MemorySafe system are used to form a context. PHOCON allows users to browse their photos as if they were exploring a concept lattice. When a user searches for photos, the matching photos are presented as a concept in a lattice. At each point in time there will be one concept which is "current". The superconcepts and subconcepts of the "current" concept are also presented for browsing. The user interface of PHOCON is illustrated in Figure 9.6. The rationales for this way of using the given screen are as follows.

- The two-dimensional space available on the screen is regarded as a scarce resource ("screen real estate"). This is because the physical screen is actually limited (although this will depend on the device) and also because of the information-processing capability of users. One possibility, which was explained in Section 9.3, would be to show the lattice itself in a line diagram, putting a set of the thumbnails of photos at each node. However, the thumbnails would then be hardly visible. So PHOCON focuses merely on the "current" concept.
- The "current" concept is shown as the set of the photos that it contains. It has half or more of the screen space in the PHOCON interface for better presentation. For users, the concept is a coherent collection of photos (he or she may think of it as an automatically composed album).
- The remaining design decision is how to show the information needed to let the user navigate through a lattice. Of the two alternatives, sets of thumbnails or sets of keywords, the latter was chosen to present the superconcepts and subconcepts of the "current" concept. The former was found to be unusable, again because the thumbnails were hardly visible. Each set of keywords, working like a button, can be used for going to that concept, as shown in Figure 9.6.

9.4.1 Browsing photos

The interaction between the program and a user starts with the program presenting all the photos in the system and their shared keywords, namely



Superconcepts
{..., -2002, -Turku, -Finland}
{..., -Summer, -School, -2002, -Turku}
{..., -School, -Turku, -Finland}

Current concept
{Summer, School, 2002, Turku, Finland}

Subconcepts
{..., +Yuechen}
{..., +Favorite}
{..., +Museum}
{..., +Castle}
{..., +Helsinki}
{..., +Stockholm}
{..., +Natali}

#object=172 #attribute=5 #Superconcept=3 #Subconcept=7

Figure 9.6. A snapshot of the PHOCON user interface for browsing.

the top concept of the context of photos. From then on, a user can use the following options to browse the photos in the system. Some options are adopted from Internet browsers.

- Browse the current concept by browsing the thumbnails of the photos in the current concept and viewing the original images of them.
- Explore any superconcept.
- Explore any subconcept.
- Go back to the last visited concept.
- Go forward to a recently visited concept.
- Go to the top concept containing all the photos.
- Search by keywords.

To ease the browsing task, visited and not-visited concepts are coded using different colors in the actual user interface. In principle, other traditional browser features, such as a history list, could be added as well.

9.4.2 Searching for photos

In searching for photos, a user specifies a list of keywords. The program presents the smallest concept containing all the keywords and its direct subconcepts and direct superconcepts to the user.

As illustrated in Figure 9.6, a user wants to find all the photos containing the keywords “2002” and “Finland” . The program finds that there are 172 photos matching this criterion. The program also discovers that those photos share some other keywords, notably “Summer”, “School” and “Turku”. Most likely, those photos were all taken during a summer school at Turku, Finland. The program presents a matching concept containing those 172 photos and the shared keywords to the user.

In order to find more specific photos, a user can use the displayed direct subconcepts, without examining the listed photos. Direct subconcepts have additional keywords other than the specified ones and can be used for refining search criteria. As illustrated in Figure 9.6, those 172 photos are grouped into seven direct subconcepts of the matching concept. For those subconcepts, only additional keywords are displayed in the interface to enable users to recognize the difference easily. This treatment is helpful when there are many subconcepts.

If there are no interesting photos, a user can relax the search criteria by dropping some keywords. This is usually done by another search procedure in other photo browsing tools. In PHOCON, the user can use the displayed superconcepts to fulfill this goal. Figure 9.6 shows that there are three superconcepts of the current matching concept. Direct superconcepts have fewer keywords than the matching concept and only the missing keywords are displayed.

9.4.3 Metadata editing

Using existing software to annotate digital photos is a cumbersome task for end users. Usually people have to do it file-by-file and directory-by-directory. To help people in annotating their digital photos, a new interface was designed, which is illustrated in Figure 9.7.

On this user interface, the user has an overview of all the photos in the system. The user can select any photo by clicking on the thumbnail of that photo. The user can also perform a multiple selection by first pressing a mouse button on the thumbnail of a photo, then holding and dragging into the thumbnail of another photo, and finally releasing the mouse button. In this way, all photos whose thumbnails are fully or partially covered by the rectangle outlined by the dragging operation are selected. There is also an option that allows the user to select photos in multiple steps. Figure 9.7 illustrates a multiple selection.

The user can edit the annotation of the selected photo(s) using the options provided in the editing panel below the thumbnail panel. The user can append typed-in keywords to the selected photo(s), remove the typed-in keywords from the annotation of the selected photo(s), and replace the keywords of the selected photo(s) with the typed-in keywords.

Using this user interface, the user need not necessarily annotate his or her digital photos file-by-file and directory-by-directory. Annotating photos becomes a rather easy and convenient activity for the user.

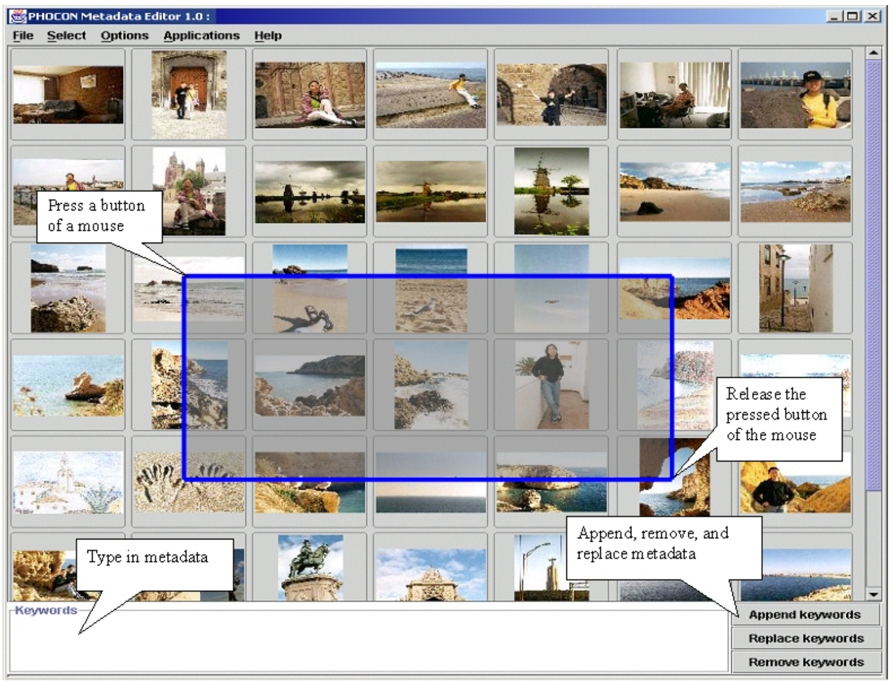

PHOCON Metadata Editor 1.0 :
File Select Options Applications Help
Press a button of a mouse
Release the pressed button of the mouse
Type in metadata
Append, remove, and replace metadata
Keywords
Append keywords
Replace keywords
Remove keywords

Figure 9.7. A snapshot of the PHOCON user interface for editing metadata.

9.5 Related work

Unlike with existing photo browsing tools, the proposal to adopt photo concepts (pairs of photo files and keywords), rather than photo files, as the central notion in the user interface of a photo browsing system is innovative. In handling search requests, moreover, PHOCON presents search results in terms of concepts and allows the user to relax or intensify searching criteria by navigation. Such a treatment of search is new too.

Many tools have been built for Concept Lattice Analysis. TOSCANA is a management system for conceptual information Systems [169]. The Formal Concept Analysis Library is an independent C++ class library which implements the basic data structures and algorithms for Formal Concept Analysis [155]. ConImp is a DOS program for contexts, concepts, concept lattices and implications [22]. Diagram is a program for drawing line diagrams of concept lattices [170]. ConExp is a Java program for various tasks of qualitative data analysis based on Formal Concept Analysis [175]. ConceptRefinery is a tool for legacy code analysis [26]. These tools are largely used in academic schools for data analysis and knowledge engineering. Most of the tools were developed as general-purpose tools and are not tuned for a specific application domain.

Formal concepts analysis has been applied in a variety of application domains. T. Rock et al. implemented a library information system based on Formal Concept Analysis [127]. R. Cole and P. Eklund applied this method in an information management system of medical records [25]. With G. Stumme, Cole built an email management system in which emails are stored and presented in a concept lattice [26]. T. Kuipers et al. combined Formal Concept Analysis and type inference in a semiautomatic approach to find objects in Cobol legacy systems [82]. P. Becker and P. Eklund proposed to use Formal Concept Analysis for document retrieval [11]. T. Tilley applied Formal Concept Analysis to software modelling [159] and formal specification visualization [160]. In those systems, line diagrams are the main entries for navigating concept lattices in data analysis. PHOCON is a new application of Formal Concept Analysis. PHOCON provides users with an overview of a photo concept of interest in a concept lattice and allows them to navigate the rest of the lattice in a step-wise manner. This method proves to be rather useful for applications with visual information.

9.6 Concluding remarks

In this chapter the design of PHOCON was described, together with its mathematical and algorithmic foundation. The user interaction and implementation of PHOCON were also explained. PHOCON is based on the idea of step-wise lattice navigation [123]. To the best of the author's knowledge, the proposal to adopt concepts (rather than photos) as the central notion in the user interface of a photo browsing system is new. Moreover, the feasibility of this proposal was demonstrated by implementing a working prototype. Several variations and combinations with traditional search methods have also been explored.

The content of this chapter was presented at the International Conference on Conceptual Structures 2003 in Dresden, Germany [123]. The presentation received very positive feedback. People from the audience kept asking where they could download this tool. They found that, for people who have large collections of digital photos, PHOCON provides a very intuitive and easy way of typing in metadata and locate images. For experts working in Formal Concept Analysis, PHOCON proves to be useful in explaining the theory to people without mathematical knowledge. Several application domains were identified as well.

- Integrate PHOCON into existing photo browsing tools, such as ACDSee.
- Develop similar tools for image analysis in pathology.
- Apply this approach to document retrieval in file systems or Internet documents retrieval.

So far, Formal Concept Analysis has been used mainly in data analysis in the fields in which "data" are just present. This is not the case with digital photos. Digital photos are usually poorly annotated with meaningful metadata. The use of Formal Concept Analysis in digital photo browsing demands metadata that describe the knowledge of people about their photos.

Photos taken by digital cameras have some built-in metadata, such as the date of shooting. In the future, location information, such as the place where a photo is taken, can be embedded in image files with help of the GPS systems. Other types of metadata, such as persons in a photo and events at the time of shooting, are not included. Feature extraction techniques can provide visual aspects of images, but not contents, semantics or emotions which viewers associate with the images.

Truly "meaningful" metadata should come from the user. On the one hand, nobody can describe the content, the meaning or the memories evoked by a photo better than the photo's owner. On the other hand, everybody has his or her own vocabulary and way of describing things. Therefore, metadata acquisition is a personal activity and the user must be involved in metadata acquisition.

Metadata do not come for free. The user must make an effort to introduce metadata in the system. Finding ways of assisting users in this activity is the key to the boom in metadata-related applications. Metadata acquisition is an interplay between the photos (the things people see) and the metadata (the terms people use to describe what they see). It is very interesting to see whether and how well Formal Concept Analysis could be applied to metadata acquisition, and not only in data analysis. PHOCON is the first attempt in this direction. It facilitates and stimulates users to input their knowledge into the system. It will be interesting to investigate other alternatives in this direction in the future.

10. Conclusions

This thesis is based on a comprehensive study of the problem of disconnected updates with the help of formal methods. Various types of conflicts were identified and classified. They are update conflicts, deletion conflicts, renaming conflicts and structure conflicts. Incautious conflict resolution may introduce update loss, file miss, identity loss or name clash. To resolve those conflicts, the identity-based history synchronization method was used in the MemorySafe system, a distributed data management system for home environments.

Distinct characteristics of home environments affect the design of the MemorySafe system. To address the heterogeneity of home environments, data identities were introduced in the MemorySafe system for modelling digital multimedia data objects that are often replicated across interconnected devices. To tackle the dynamics of the home environments, characteristic-entry logs were deployed to record disconnected updates in the MemorySafe system, enabling less resource consumption and faster data synchronization. To deal with the user-centric nature of home environments, semantic rules were used to reduce user involvement in data synchronization.

This thesis described how characteristic-entry logs were implemented and used in the data synchronization of the Memory system. A case study revealed the practical use of such logs in real environments: using characteristic-entry logs would reduce the amount of system resources such as memory, disk storage, and network bandwidth used for storing and exchanging logs to as little as 72% of the amount of system resources required when using normal logs. In the case of more specific types of data, such as directories, configuration files and textual files, using characteristic-entry logs would reduce resource use to 44%, 43%, and 33%, respectively. This performance gain is typically useful for portable devices with memory and bandwidth constraints and helps to speed up the synchronization process.

Several applications built on top of the MemorySafe system were presented in this thesis to illustrate how the system is used in practice. Among them, in particular, photo concept browsing is a direct application of the mathematical theory of Formal Concept Analysis, which demonstrates the power of formalism in user-centric applications.

In this thesis, data synchronization was studied using formal methods. First, disconnected updates were defined and analyzed using the Z notation. The formal definitions of disconnected updates provided precision in conflict classification. This formal treatment helped the author in understanding the problem and will also be beneficial to other researchers in the field of data synchronization. Second, a formal model of characteristic-entry logs was developed. In this model how characteristic-entry logs could be constructed and how they could be converted from normal logs were defined in a mathematical manner. This result illustrates that in practice characteristic-entry logs can be implemented along with normal logs in real systems and they can alternatively be converted from normal logs on the fly for the purpose of data synchronization only. As a result, the process of data synchronization can be sped up thanks to using characteristic-entry logs while system traceability can be retained because of using normal logs. Finally, the soundness of using characteristic-entry logs in data synchronization was proven. It was rigorously proven that the result obtained by using characteristic-entry logs in semantic data synchronization is the same as that obtained by using normal logs. Characteristic-entry logs and normal logs can hence be used interchangeably in semantic data synchronization. This result provides a formal justification for using characteristic-entry logs in data synchronization in real systems.

Postscript

The work presented in this thesis was carried out in the Phenom project of Philips Research, the Netherlands. Phenom focused on user-system interaction in future home environments. This type of research is quite different from any traditional research that is carried out in computing science, electronic engineering or user-system interaction only. Moreover, doing research in industrial institutions has its own characteristics. Therefore, it is worthwhile to reflect on the design process of the MemorySafe system.

Designing systems for home environments. In designing a system for home environments, many user requirements are not technical. Instead, they concern environments and end users. For example, home appliances are made by different vendors and various standards are in use. Any system for home use should take into account the heterogeneous nature of home environments. Devices in home environments can be switched on or off at any time. Any system for home environments should behave correctly in such a dynamic context. End-users are not assumed to have prerequisite knowledge of the technical specifications of devices or systems. Administrative efforts should not be required, whenever possible. The intermingling of non-technical constraints challenges any system design in home environments. Academic research often tends to focus on functional advancement, regardless of non-technical constraints such as plug-n-play and ease-of-use.

Several distinct non-technical requirements of home environments were taken into account in the design of the MemorySafe system. For example, data identities are introduced and preserved across devices in the system. This mechanism places a minimal requirement for locating replicated data objects on in-home devices, addressing the heterogeneity of the devices in home environments. Due to the dynamics of home environments, a fast data synchronization mechanism using characteristic-entry logs was implemented in MemorySafe to handle disconnected updates. To reduce end-users' involvement, semantic rules are used to automate conflict resolution in data synchronization.

Designing a distributed data management system for home environments which will be highly appreciated by end users requires a joint effort of scientists and researchers from multiple disciplines. Traditionally, such tasks are the privileges of computing scientists and engineers. As interconnected devices are emerging in home environments, in-depth understanding of the interaction between human beings and the "cyberspace" that they share, especially in home environments, is needed. A system design without a decent treatment of end-users' needs will not be appreciated by the end users.

The design of the MemorySafe system benefited from studies on user-system interactions in home environments. For example, location transparency is one of key features implemented in many existing distributed systems. In the design of the MemorySafe system, however, it was discovered in a user study that hiding the information on where data are stored makes some end users hard to manage their data. Many home users are in favor of "devices being in control" and would like to have more control on what data should be stored and where. Therefore, in its final design, MemorySafe allows both location-aware and location-transparent data accesses to satisfy different requirements.

In home environments, there is a wide spectrum of new research challenges for system design in the convergence of distributed computing and user-system interactions. The interplay between end users and intelligent systems will also boost new industrial system design.

Doing research in industrial institutions. Doing research in industrial institutions requires contributions to system prototyping and implementation. Design, prototyping, demonstration and evaluation are constantly recurring features in the design process. A good balance between prototyping and scientific research is required.

On the one hand, the prototyping obligation sometimes consumes too much time and effort, especially before major research exhibition events. Scientific research requires a kind of abstraction that is contradictory to material implementation. Overdoing prototyping may introduce more problems rather than yielding good results. On the other hand, prototyping activities provide opportunities for obtaining a better understanding of research problems and discovering research directions. For example, at the very beginning of the Phenom project it was thought that data synchronization would be common to all data objects. After the case study on the use of the MemorySafe system, it was found that directories, configurations and settings are indeed much more frequently updated than multimedia objects. This finding implies that research effort should focus on distributed profile management and synchronization in the future.

Cyclic design process. The MemorySafe system experienced two design cycles in the four years' research period, keeping pace with the Phenom project. (Phenom also had two design phases, each of them involving one design cycle.) Each design cycle comprises design, prototyping and evaluation. This cyclic development helped in avoiding major design flaws in the final deliverables.

In the first evaluation of the MemorySafe system, performance was found to be an issue for tasks like the Browsing Assistant collecting metadata from all the photos in MemorySafe. To address this problem, indexing was later implemented to improve system performance. Without the early testing and evaluation, the performance problem would not have been discovered until the final integration of the Phenom demonstration systems, which would have severely delayed the progress of the whole Phenom project. Therefore, it is always wise to fully test the key components of a large system design before system integration. In a full run design project, a short design cycle can be used for this purpose.

Applying formal methods in system design. Formal methods and theories were used on several occasions throughout the design of the MemorySafe system: formal definition and analysis of disconnected updates, formal system specification, modelling of characteristic-entry logs, formalization of semantic rules and data synchronization, and application development of photo concept browsing.

The actual realization of those formally developed designs was actually only a small part of the overall implementation of the MemorySafe system. This is quite common in practice. It would be infeasible to "do everything formally" in real system design. Formalization efforts should therefore always focus on selected topics, usually system modelling and specification. In the design of the MemorySafe system, logging systems and semantic data synchronization were treated formally, and the design of the system is consequently distinct from that of many existing systems.

Bibliography

1. Aarts, E., Harwig, R., Schuurmans, M.: Ambient Intelligence. In Denning, P.J. (Ed.): The Invisible future: the seamless integration of technology in everyday life. McGraw-Hill (2002) 235–250
2. ACD Systems: ACDSee. http://www.acdsystems.com/
3. Adar, E., Huberman, B.A.: Free riding on Gnutella. First Monday 5, 10, 1998.
4. Afonso, A.P., Silva, M.J., Campos, J.P., Regateiro, F.S.: The design and implementation of the Ubidata information dissemination framework. In Proc. First International Symposium on Handheld and Ubiquitous Computing (HCI)'99. LNCS 1707. Spring-Verlag, 1999, 371–373.
5. Ahamad, M., Bazzi, R.A., John, R., Kohli, P., Neiger, G.: The Power of Processor Consistency. Technical Report GIT–CC–92/34, College of Computing, Georgia Institute of Technology, March 1993.
6. Anderson, J.R., Bower, G.H. (Eds.): Human Associative Memory: A Brief Edition. The Experimental Psychology Series. Lawrence Erlbaum Associates Publishers (1980)
7. Andrews, D., Ince, D.:Practical Formal Methods with VDM, McGraw Hill, September 1991.
8. Baeten, J.C.M., Weijland, W.P.: Process algebra. Cambridge University Press, Cambridge tracts in theoretical computer science 18, 1990.
9. Balabanovic, M., Chu, L.L., Wolff, G.J.: Storytelling with digital photographs. In Proceedings of CHI 2000. (2000) 564–571
10. Balasubramaniam, S., Pierce, B.C.: What is a File Synchronizer? CSCI Technical Report #507. Indiana University (1998)
11. Becker, P., Eklund, P.: Prospects for document retrieval using formal concept analysis. In Proceedings of Sixth Australian Document Computing Symposium. Australia (2001) 5–9
12. Bederson B.B.: PhotoMesa: A Zoomable Image Browser Using Quantum Treemaps and Bubblemaps. In Proceedings of UIST 2001, ACM Symposium on User Interface Software and Technology. CHI Letters, 3(2) (2001) 71–80
13. Bergstra, J.A., Heering, J., Klint, P.: Module algebra. Journal of ACM, 37(2), April 1990, 335–372.
14. Berners-Lee, T.: Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web. Network Working Group RFC 1630. (1994)
15. Berners-Lee, T., Fielding, R., Masinter, L.: Uniform Resource Identifiers (URI): Generic Syntax. Network Working Group RFC 2396. (1998)
16. Berners-Lee, T.: Masinter, L., McCahill, M.: Uniform Resource Locators (URL). Network Working Group RFC 1738. (1994)
17. Bernstein, P.A., Goodman, N.: Timestamp-based algorithms for concurrency control in distributed database systems. In Proc. 6th International Conference on Very Large Data Bases, Canada, (1980).

18. Bernstein, P.A., Goodman, N.: An algorithm for concurrency control and recovery in replicated distributed databases. ACM Transactions on Database Systems, Vol. 8, No. 4, December 1984, 596–615
19. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency control and recovery in database systems. Addison Wesley, 1987.
20. Berthold, H.: Physical and logical integration of data and media objects. “Föderierte Datenbanken” workshop . Shaker Verlag. 1998. 143–161.
21. Braam, P.J.: The Coda distributed file system. Linux Journal(50). (1998)
22. Burmeister, P.: ComImp. http://www.mathematik.tu-darmstadt.de/ags/ag1/Software/DOS-Programme/Welcome_en.html
23. Cellary, W., Gelenbe, E., Morzy, T.: Concurrency Control in Distributed Database Systems. Noord-Holland, 1988.
24. Clarke,I., Sandberg, O., Wiley, B., Hong, T.: Freenet: A distributed anonymous information storage and retrieval system. In Proceedings of ICSI Workshop on Design Issues in Anonymity and Unobervability. Berkeley, California. 2000.
25. Cole, R., Eklund, P.: Scalability in Formal Concept Analysis: A Case Study using Medical Texts. Computational Intelligence, Vol. 15, No. 1. (1999) 11–27
26. Cole, R., Stumme, G.: CEM - a conceptual email manager. In Mineau, G., Ganter, B. (Eds.): International Conference on Conceptual Structures. Lecture Notes in Computer Science, Vol. 1867. Springer-Verlag, (2000) 438–452
27. Conduits Technologies, Inc.: Peacemaker Pro. http://www.conduits.com/
28. Date, C.J.: An Introduction to Database Systems. Addison Wesley Longman, Inc., 2000.
29. Davidson, S.B. , Garcia-Molina, H., Skeen, D.: Consistency in Partitioned Networks, *ACM Computing Surveys*, 17(3), 1985, 341–370.
30. de Jong, N.: Phenom Empire Architecture. Technical Note PR-TN-2003/0787. Koninklijke Philips Electronics N.V.
31. Demers, A., Petersen, K., Spreitzer, M., Terry, D., Theimer, M., Welch, B.: The Bayou Architecture: Support for Data Sharing among Mobile Users. In Proc. IEEE workshop on mobile computer systems and applications. 1994, 2–7.
32. Dijk, E.O.: In-home location systems. Ph.D. thesis. Eindhoven University of Technology. (To be published).
33. Dijk, E.O., van den Hoven, E.A.W.H., van Loenen, E.J., Qian, Y., Tedd, D.N., Teixeira, D.: A Portable Ambient Intelligent Photo Browser. Nat.Lab. Technical Note NL-TN 2000/257. Koninklijke Philips Electronics N.V. (2000)
34. Dijk, E.O., de Jong, N., van den Hoven, E.A.W.H., van Loenen, E.J., Qian, Y., Teixeira, D.: Phenom Scenarios. Nat.Lab. Technical Note NL-TN 2001/336. Koninklijke Philips Electronics N.V. (2001)
35. Dijkstra, E.W., Scholten, C.: Predicate calculus and program semantics. Springer Verlag, 1990.
36. Diller A.: Z: An Introduction to Formal Methods. John Wiley and Sons. 1990.
37. Dwyer, D., Bharghavan, V.: A mobility-aware file system for partially connected operation. In ACM Operating Systems Review, 31(1), (1997) 24–30.
38. Ebling, M.R.: Translucent Cache Management for Mobile Computing. Ph.D. Thesis CMU-CS-98-116, Carnegie-Mellon University, Pittsburgh PA US, March 1998.
39. Eberhard, J., Tripathi, A.: Efficient Object cahing for distributed Java RMI applications. In Guerraoui, R. (Ed.): Middleware 2001, LNCS 2218. Springer-Verlag. (2001) 15–35.
40. E-Book Systems, Inc.: FlipAlbum. http://www.flipalbum.com/

41. Edwards, W.K., Grinter, R.E.: At home with ubiquitous computing: seven challenges. Ubicomp 2001. (2001) 256–272.
42. Feijs, L.M.G., Jonkers, H.B.M.: Formal specification and design. Cambridge University Press. Cambridge tracts in theoretical computer science 35, 1992.
43. Feijs, L.M.G., Jonkers, H.B.M., Middelburg, C.A.: Notations for software design. Springer-Verlag, 1994.
44. Feijs, L.M.G., Qian, Y.: Component algebra. Science of Computer Programming 42(2002) 173–228.
45. Ferreira, J.M.A.: Browsing Assistant. Nat.Lab. Technical Note NL-TN 2002/284. Koninklijke Philips Electronics N.V. (2002)
46. Fitzpartrick, T., Gallop, J., Blair, G., Cooper, C., Coulson, G., Duce, D., Johnson, J.: Design and application of TOAST: an adaptive distributed multimedia middleware platform. IDMS 2001. (2001) 111–123.
47. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer Verlag (1999)
48. Gifford, D.K.: Weighted voting for replicated data. Proc. Seventh Symp. on Operating Systems Principles. ACM, (1979) 150–162.
49. Gifford, D.K., Jouvelot, P., Sheldon, M.A., O'Toole, J.W.Jr.: Semantic File Systems. In Proceedings of the 13th ACM Symposium on Operating Systems Principles. October 1991, 16–25.
50. Gracenote: Gracenote. http://www.gracenote.com/
51. Guy, R.G., Heidemann, J.S., Mak, W., Page Jr., T.W., Popek, G.J., Rothmeier, D.: Implementation of the Ficus Replicated File System. Proceedings of the Summer USENIX Conference 1990. (1990) 63–71.
52. Guy, R.G., Popek, G.J. Page, Jr.T.W.: Consistency algorithms for optimistic replication. In Proc. the First International Conference on Network Protocols. IEEE. 1993.
53. Guy, R., Reiher, P., Ratner, D., Gunter, M., Ma, W., Popek, G.: Rumor: mobile data access through optimistic peer-to-peer replication. In Proc. ER'98 Workshop on Mobile Data Access, 1998. 254–265.
54. Hand Raisers, Inc.: SyncTalk. http://www.synctalk.com/
55. Harper, R.F., Honsell, F., Plotkin, G.: A Framework for Defining Logics. Journal of the ACM 40(1), 1993, 143–184.
56. Heidemann, J., Goel, A., Popek, G.: Defining and measuring conflicts in optimistic replication. Technical report UCLA-CSD-950033, University of California, Los Angeles, 1995.
57. Herlihy, M., Wing, J.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Prog. Lang. Syst. 12(3) 1991, 463–492.
58. Hoare, C.A.R.: Communicating sequential processes. Prentice Hall International, 1985.
59. Holmquist, L.E., Redströ, Ljungstrand P.: Token-based access to digital information. In Proc. First International Symposium on Handheld and Ubiquitous Computing (HCI)'99. LNCS 1707. Spring-Verlag, 1999, 234–245.
60. Holzmann, G.J.: Design and Validation of Computer Protocols. Prentice Hall, 1990.
61. Howard, J.H.: An overview of the Andrew file system. In Proceedings of the Winter USENIX Conference. (1988) 23–26.
62. Huizinga, D.M., Heflinger, K.A.: Two-level client caching and disconnected operation of notebook computers in distributed systems. In Proceedings of the 1995 ACM Symposium on Applied Computing. 1995. 390–395.
63. Huizinga, D.M., Mann, P.: Disconnected operation for heterogeneous servers. In Proceedings of the 1996 ACM Symposium on Applied Computing. 1996. 312–321.

64. Huizinga, D.M., Sherman, H.: File hoarding under NFS and Linux. In Proceedings of the 1998 ACM Symposium on Applied Computing. 1998. 409–415.
65. Huston, L.B., Honeyman, P.: Disconnected operation for AFS. In Proceedings of the USENIX Symposium on Mobile and Location-Independent Computing, USENIX (1993) 1–10.
66. Huston, L.B., Honeyman, P.: Partially connected operation. In Proceedings of the USENIX Symposium on Mobile and Location-Independent Computing, USENIX (1995) 91–97.
67. Huston, L.B., Honeyman, P.: Peephole log optimization. CITI Technical Report 95-3, University of Michigan Ann Arbor. 1995.
68. Internet Movie Database Inc.: The Internet Movie Database. http://www.imdb.com/
69. Iomega Corporation: QuikSync. http://www.iomega.com/
70. Joseph, A.D., de Lespinasse, A.F., Tauber, J.A., Gifford, D.K., Kaashoek, M.F.: Rover: a toolkit for mobile information access. In Proceedings of the 15th ACM Symposium on Operating Systems Principles, 1995, 156–171.
71. Joslyn, C.: Semantic Webs: A Cyberspatial Representational Form for Cybernetics. In Proceedings of Cybernetics and Systems '96, Austrian, 1996, 905–910.
72. Kang, H., Shneiderman, B.: Visualization Methods for Personal Photo Collections: Browsing and Searching in the PhotoFinder. In Proceedings of the IEEE International Conference on Multimedia and Expo (III). (2000) 1539–1542.
73. Kiciman, E., Fox, A.: Using dynamic mediation to integrate COTS entities in a ubiquitous computing environment. In Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing (HUC2K), LNCS 1927. Springer-Verlag. (2000) 211–226.
74. Kistler, J.J., Satyanarayanan, M.: Disconnected Operation in the Coda File System, *Proc. 13th ACM Symposium on Operating Systems Principles*, 25(5), Pacific Grove, U.S., 1991, 213–225.
75. Kohonen, T.: Self-Organization and Associative Memory. Springer Series in Information Sciences, Vol. 8. Springer-Verlag, Berlin Heidelberg New York Tokyo (1984)
76. Koninkijke Philips Electronics N.V.: The LiMe project. http://www.design.philips.com/lime
77. Koninkijke Philips Electronics N.V.: The "Perceptive Home Environments" project. http://www.project-phenom.info/
78. Kubiatowicz, J.: Extracting guarantees from chaos. Communications of the ACM. 46(2), February 2003, 33–38.
79. Kuenning, G.H., Popek, G.J., Reiher, P.: An analysis of trace data for predictive file caching in mobile computing. In Proceedings of the USENIX Summer Conference, (1994) 291–303.
80. Kuenning, G.H.: The design of the Seer predictive caching system. In IEEE Workshop on Mobile Computing Systems and Applications, 1994.
81. Kuenning, G.H., Popek, G.J.: Automated Hoarding for Mobile Computers. In Proceedings of the 16th ACM Symposium on Operating Systems Principles, (1997) 264–275.
82. Kuipers, T., Moonen, L.: Types and Concept Analysis for Legacy Systems. In Proceedings of the IEEE International Workshop on Program Comprehension. (2000) 221–230.
83. Kumar, P., Satyanarayanan, M.: Flexible and Safe Resolution of File Conflicts. In Proc. USENIX Winter 1995 Conference on Unix and Advanced Computing Systems. (1995) 95–106.

84. Kumar, P., Satyanarayanan, M.: Log-Based Directory Resolution in the Coda File System. Proceedings of the Second International Conference on Parallel and Distributed Information Systems. (1993) 202-213.
85. Kung, H.T., Robinson, J.T.: On optimistic methods of concurrency control. ACM Transactions on Database Systems. 6(2), (1981) 213-226.
86. Kuznetsov, S.O., Obëdkov, S.A.: Comparing Performance of Algorithms for Generating Concept Lattices. In Workshop of ICCS 2001 on Concept Lattice-based theory, methods, and tools for Knowledge Discovery in Databases. http://www.lattices.org/Doc/paper4_kuznetsov.pdf
87. Ladin, R., Liskov, B., Shrira, L., Ghemawat, S.: Providing high availability using lazy replication. ACM Transaction on Computer Systems, 10(4), 1992, 360-391.
88. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Communication of ACM, vol. 22 (1978) 558–564.
89. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A Notation for Detailed Design. In Kilov, H., Rumpe, B., Simmonds, I. (Eds.): Behavioral Specifications of Businesses and Systems. Kluwer, 1999, 175–188.
90. Lee, J.: An end-user perspective on file sharing systems. Communications of the ACM. 46(2), February 2003, 49–53.
91. Levy, E., Silberschatz, A.: Distributed File Systems: Concepts and Examples. ACM Computing Surveys, Vol.22, No. 4, 1990, 321–374.
92. Lindig, C.: Fast concept analysis. In Stumme, G.(Ed.): Working with Conceptual Structures - Contributions to ICCS 2000. Shaker-Verlag (2000) 151–161
93. Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., Shrira, L., Williams, M.: Replication in the Harp file system. In Proc. of the 13th ACM Symposium on Operating Systems Principles. 1991. 226–238.
94. Loh, Y.H., Hara, T., Tsukamoto, M. Nishio, S.: A hybrid method for concurrent updates on disconnected databases in mobile computing environments. In Proceedings of the 2000 ACM Symposium on Applied Computing. 2000. 563–565.
95. Microsoft Corporation: The Distributed Component Object Model (DCOM). http://www.microsoft.com/com/tech/dcom.asp
96. Microsoft Corporation: The Component Object Model Specification, Version 0.9. Microsoft Corp. (1995)
97. Microsoft Corporation: Using the Unique Identifier Data Type. http://msdn.microsoft.com/library/psdk/sql/r_model_20.htm. Microsoft Corp.
98. Milner, R.: Communicating and Mobile Systems: the Pi-Calculus, Cambridge University Press, 1999.
99. Mills, D.L.: Measured performance of the Network Time Protocol in the Internet system. ACM Computer Communication Review 20, 1 (January 1990), 65–75.
100. Mills, D.L.: Improved algorithms for synchronizing computer network clocks. IEEETNWKG: IEEE/ACM Transactions on Networking 3(3), 1995, 245–254.
101. Mills, T.J., Pye, D., Sinclair, D., Wood, K.R.: Shoebox: a digital photo management system. http://citeseer.nj.nec.com/379884.html
102. Moats, R.: URN Syntax. Network Working Group RFC 2141. (1997)
103. Mummert, L.B., Ebling, M.R., Satyanarayanan, M.: Exploiting weak connectivity for mobile file access. In Proc. 15th ACM Symposium on Operating Systems Principles. 1995.
104. Necula, G.C.: Proof-carrying code design and implementation. In Schwichtenberg, H., Steinbrüggen (Eds.): Proceedings of the NATO Advanced Study

Institute on Proof and System Reliability, Marktoberdorf, Germany, (2002) 261–288.

105. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL – a proof assistant for higher-order logic. LNCS 2283.

106. OASIS, the XML interoperability consortium: The Extensible Markup Language (XML). http://www.xml.org/

107. On, G., Schmitt, J.B., Steinmetz, R.: Design and implementation of a QoS-aware replication mechanism for a distributed multimedia system. In Proceedings of IDMS 2001, LNCS 2158, Springer-Verlag, (2001) 38–49.

108. OSMB, LLC: Gnutella. http://gnutella.wego.com/

109. Parker, D.S., Popek, G.J., Rudisin, G., Stoughton, A., Walker, B., Walton, E., Chow, J., Edwards, D., Kieser, S., Kline, C.: Detection of Mutual Inconsistency in Distributed Systems. IEEE Transactions on Software Engineering 9, 3 1983(May)

110. Petersen, K, Spreitzer, M.J., Terry, D.B.: Flexible update propagation for weakly consistent replication. In Proc. 16th ACM Symposium on Operating Systems Principles. (1997) 288–301.

111. Phatak, S.H., Badrinath, B.R.: Data partitioning for disconnected client server databases. In Proc. the ACM International Workshop on Data Engineering for Wireless and Mobile Access. (1999) 102–109.

112. Pierce, B.C.: Unison File Synchronizer. http://www.cis.upenn.edu/~bcpierce/unison/index.html

113. Pierce, B.C., Vouillon, J.: How to Specify a File Synchronizer. http://www.pps.jussieu.fr/~vouillon/publi.html#unisonspec

114. Pierre, G., van Steen, M.: Globule: a platform for selfreplicating Web documents. In Proceedings of the 6th International Conference on Protocols for Multimedia Systems, LNCS 2213, (2001) 1–11.

115. Pierre, P., Kuz, I., van Steen, M., Tanenbaum, A.S.: Differentiated Strategies for Replicating Web documents. In Computer Communications 24(2), (2001), 232-240.

116. Pitoura, E., Bhargava, B.: Data Consistency in Intermittently Connected Distributed Systems. Knowledge and Data Engineering 11(6), 1999, 896–915.

117. Popek, G.J., Guy, R.G., Page Jr., T.W., Heidemann, J.S.: Replication in Ficus Distributed File Systems. Proceedings of the Workshop on Management of Replicated Data 1990. (1990) 20–25.

118. Pu, C., Leff, A.: Replica control in distributed systems: an asynchronous approach. In Proc. of ACM SIGMOD Int'l Conf. on Management of Data, 1991. 377–386.

119. Pumatech, Inc.: IntelliSync. http://www.pumatech.com/

120. Qian, Y., Udink, R., Feijs, L.M.G.: Data synchronization in mobile and ubiquitous computing environments. Nat.Lab. Manuscript NL-MS 2001/100. Koninklijke Philips Electronics N.V. (2001)

121. Qian, Y., Udink, R., Feijs, L.M.G.: A photo management system for future home environments. In Proceedings of International ITEA Workshop on Virtual Home Environments. Paderborn, Germany. Shaker Verlag (2002) 93–101.

122. Qian, Y., Feijs, L.M.G, Udink, R.: Characteristic-entry logs in the MemorySafe Information System. In Proceedings of the 14th IASTED International Conference on Parallel and Distributed Computing and Systems. MIT, Cambridge, USA. ACTA Press. 2002, 185–190.

123. Qian, Y., Feijs, L.M.G, : Stepwise Concept Navigation. In de Moor, A., Ganter, B. (Eds.): Using Conceptual Structures: Contributions to International Conference on Conceptual Structures 2003. Dresden, Germany. Shaker-Verlag (2003) 255–268.

124. Qian, Y.: Formal development of a distributed logging mechanism supporting disconnected updates. In Dong, J.S., Woodcock, J. (Eds.): Proceedings of International Conference on Formal Engineering Methods 2003 (ICFEM 2003). Singapore. LNCS 2885. Springer-Verlag (2003) 338–358.
125. Ramsey, N., Csirmaz, E.: An algebraic approach to file synchronization. Technical Report TR-05-01, Harvard University Dept. of Computer Science, Cambridge MA (USA), May 2001.
126. Reiher, P., Heidemann, J., Ratner, D., Skinner, G., Popek, G.: Resolving File Conflicts in the Ficus File System. Proceedings of the Summer USENIX Conference 1994. (1994) 183–195.
127. Rock, T., Wille, R.: Ein TOSCANA-System zur Literatursuche. In: Stumme G., Wille R. (Eds.): Begriffiche Wissensverarbeitung: Methoden und Anwendungen. Springer, Berlin-Heidelberg (2000) 239–253
128. Rodden, K., Wood, K.: How do People Manage Their Digital Photographs? In Proceedings of ACM CHI 2003. (2003) 409–416
129. Rodden, K., Basalaj, W.: Does organization by similarity assist image browsing? In Proceedings of CHI 2001. (2001) 190–197
130. Ross, K.A., Wright, C.R.B.: Discrete mathematics (2nd ed.). Prentice Hall, 1988.
131. Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In Proceedings of the 19th IFIP/ACM Int'l Conf. on Distributed Systems Platforms, 2001.
132. Satyanarayanan, M., Kistler, J.J., Mummert, L.B., Ebling, M.R., Kumar, P., Lu, Q.: Experience with Disconnected Operation in a Mobile Computing Environment. Proceedings of the 1993 USENIX Symposium on Mobile and Location-Independent Computing. (1993)
133. Saito, Y.: Consistency management in optimistic replication algorithms. citeseer.nj.nec.com/saito01consistency.html
134. Shankar, N.: PVS: Combining specification, proof checking and model checking. In Proceedings of FMCAD'96, LNCS 1166.
135. Shapiro, M., Rowstron, A., Kermarrec, A.M.: Application-independent reconciliation for nomadic applications. In Proc. of the ACM SIGOPS European Workshop: "Beyond the PC: New Challengers for the Operating Systems". 2000.
136. Sharman Networks Ltd: Kazaa. http://www.kazaa.com/
137. Shields, T: What's GUID.org? http://www.guid.org/
138. Silva, M., Afonso, A.P.: Designing information appliances using a resource replication model. In Proc. First International Symposium on Handheld and Ubiquitous Computing (HCI)'99. LNCS 1707. Springer-Verlag, 1999, 150–157.
139. Sollins, K., Masinter, L.: Functional Requirements for Uniform Resource Names. Network Working Group RFC 1737. (1994)
140. Spivey, J.M.: The Z Notation: a reference manual: 2nd edition. Prentice Hall. 1992.
141. Spivey, J.M.: Understanding Z: a specification language and its formal semantics. Cambridge University Press. 1989.
142. Sterbenz, J.P.G., Saxena, T., Krishnan, R.: Latency-Aware Information Access with User-Directed Fetch Behavior for Weakly-Connected Mobile Wireless Clients. BBN Technical Report 8340, BBN Technologies, May 9, 2002.
143. Sun Microsystems Incorporation: Jini Technology 1.0 API Documentation. Sun Microsystems Inc. (1999)
144. Sun Microsystems, Inc.: The Java Technology, http://www.sun.com/java/
145. Sun Microsystems, Inc.: The Jini Network Technology, http://www.sun.com/jini/

146. Swierk, E., Kiciman, E., Laviano, V., Baker, M.: The Roma Personal Metadata Service. Proceedings of the 3rd IEEE Workshop on Mobile Computing Systems and Applications. (2000)
147. SyncML Initiative Ltd.: SyncML Representation Protocol (version 1.1). http://www.syncml.org/
148. Tait, C, Lei, H., Acharya, S., Chang, H.: Intelligent file hoarding for mobile computers. In Proceedings of the first International conference on Mobile Computing and Networking. ACM, 1995, 119–125.
149. Takahashi, N., Wakita, Y., Ouchi, S., Kunieda, T.: Video content management using logical content. In Proceedings of IDMS 2001, LNCS 2158, Springer-Verlag, (2001) 193–198.
150. Tanenbaum, A.S.: Distributed Operating Systems. Prentice Hall, Inc., 1993.
151. Tanenbaum, A.S.: Computer Networks (3rd Edition). Prentice Hall, Inc., 1995
152. Tanenbaum, A.S., van Steen, M.: Distributed systems principles and paradigms. Prentice Hall, Inc. 2002
153. Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J. Spreitzer, M.J., Hauser, C.H.: Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System, *Proc. 15th Symposium on Operating Systems Principles*, Colorado, 1995, 172–183.
154. Teixeira, D.: Conversational search. Ph.D. thesis. Eindhoven University of Technology. (To be published).
155. The Formal Concept Analysis Library. http://www.mathematik.tu-darmstadt.de/ags/ag1/ Software/Library/Welcome_en.html
156. The Home Audio Video Interoperability (HAVi) Organization: HAVi, the A/V digital network resolution. http://www.havi.org/
157. The Open Services Gateway Initiatives (OSGi) Alliance: OSGi Service Platform Release 2 Specification. http://www.osgi.org/
158. The Universal Plug and Play (UPnP) Forum: Understanding UPnP: A White Paper. http://www.upnp.org/
159. Tilley, T., Hesse, W., Duke, R.: A Software Modelling Exercise Using FCA. In de Moor, A., Ganter, B. (Eds.): Using Conceptual Structures: Contributions to International Conference on Conceptual Structures 2003, Dresden, Germany. Shaker-Verlag (2003) 213–226
160. Tilley, T.: Towards an FCA based tool for visualizing Formal Specifications. In de Moor, A., Ganter, B. (Eds.): Using Conceptual Structures: Contributions to International Conference on Conceptual Structures 2003, Dresden, Germany. Shaker-Verlag (2003) 227–240.
161. Ulead Systems, Inc.: Ulead Photo Explorer. http://www.ulead.com/
162. van den Berg, J. , Jacobs, B., Poll, K.: Formal Specification and Verification of JavaCard's Application Identifier Class. In: Attali, I., Jensen, Th. (Eds.): Java on Smart Cards: Programming and Security (Springer LNCS 2041, 2001) 137–150.
163. van den Brand, M.G.J., van Deursen, A., Heering, J., de Jong, H.A., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P.A., Scheerder, J., Vinju, J.J., Visser, E., and Visser, J.: The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In Wilhelm, R. (Ed.): Compiler Construction 2001 (CC'01). Springer-Verlag. 365–370.
164. van den Hoven, E.A.W.H.: Exploring Graspable Cues of Everyday Recollections. Ph.D. thesis. Eindhoven University of Technology. (To be published).
165. van der Hoven, E.A.W.H., Dijk, E.O., van Loenen, E.J., Qian, Y., Tedd, D.N., Teixeira, D.: A tangible user interface for an ambient-intelligent photo browser. In Proceedings of the Philips User Interface 2000 Conference.

166. van Loenen, E.J., de Jong, N., Dijk, E., van den Hoven, E., Qian, Y., Teixeira, D.: Phenom. In Aarts, E., Marzano, S. (Eds.): The New Everyday, Views on Ambient Intelligence. 010 Publishers. The Netherlands. (2003) 302–303.
167. van Renesse, R., Tanenbaum, A.S.: Voting with ghosts. Proc. 8th Int'l Conf. on Distributed Computer Systems, IEEE, 1988.
168. Villate, Y., Illarramendi, A., Pitoura, E.: Keep your data safe and available while roaming. Mobile Networks and Applications, Kluwer, 2002, 315–328.
169. Vogt, F., Wille, R.: TOSCANA - A graphical tool for analyzing and exploring data. In: Tamassia, R., Tollis, I.G. (Eds.): Graph Drawing. LNCS 894. Springer-Verlag, Berlin-Heidelberg (1995) 226–233.
170. Vogt, F.: Diagram. http://www.mathematik.tu-darmstadt.de/ags/ag1/Software/DOS-Programme/Welcome_en.html
171. Weinstein, M.J., Page, T.W. Jr., Livezey, B.K., Popek, G.J.: Transactions and synchronization in a distributed operating system. In Proc. 10th Symposium on Operating Systems Principles ACM. 1985.
172. Weiser, M.: The computer for the 21st century. Scientific American, 265(3) (1991) 94–104.
173. Wille, R.: Restructuring lattice theory: an approach based on hierarchies of concepts. In: Rival, I. (Ed.): Ordered sets. Reidel, Dordrecht-Boston (1982) 445–470
174. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice Hall. 1996.
175. Yevtushenko, S.: ConExp. http://www.mathematik.tu-darmstadt.de/ags/ag1/Software/ConExp/index.html
176. Yu, H., Vahdat, A.: Building replicated Internet services using TACT: a toolkit for tunable availability and consistency tradeoffs. In Proc. of Workshop on Advanced Issues of E-commerce and Web-based Information Systems. 2000. 75–84.

Index

Summary

Thanks to the rapid advance of network technologies and device miniaturization, interconnected consumer electronic devices are appearing in home environments. As a variety of storage devices and services are appearing in home environments, there is a rapidly growing need for systems that help people easily manage their data. This thesis presents the design of MemorySafe, a distributed data management system for home environments. Maintaining consistency of data copies that are subject to disconnected updates, modifications that are made when devices are disconnected, is a problem that people repeatedly encounter in using portable devices. One design objective of the MemorySafe system is the investigation of data synchronization methods for home environments.

In the first part of this thesis heterogeneity, dynamics and ease-of-use are identified as the characteristics of home environments that affect the design of in-home data management systems. Next, an extensive survey of existing systems and tools regarding disconnected updates is presented. Most of the systems focusing on professional environments do not address the characteristics of home environments. In addition, those systems are designed in an empirical manner, without rigorous treatment. After that, several use scenarios involving disconnected updates are presented, based on which a comprehensive study of various types and consequences of disconnected updates is conducted with the help of the Z notation. It is concluded that identity-based history synchronization is suitable for in-home data management systems.

In the second part of this thesis a formal model of a system supporting disconnected updates is described using the Z notation. First, system specification and consistency requirements are formally described. Next, logs, which contain histories of updates, are formalized and their properties are investigated. After that, a new logging mechanism is introduced, namely characteristic-entry logs. A formal model of characteristic-entry logs is presented. It is also rigorously proven that characteristic-entry logs can be used in semantic data synchronization in the same way as normal logs. Such formal treatment of characteristic-entry logs justifies their use in data synchronization in practical systems.

In the third part of this thesis the design and implementation of the MemorySafe system is presented. To address the heterogeneity of home environ-

ments, data identities are introduced for modelling digital multimedia data objects that are often replicated across interconnected devices. To tackle the dynamics of the home environments, characteristic-entry logs are deployed to record disconnected updates in the MemorySafe system, leading to less resource consumption and faster data synchronization. To deal with the user-centric nature of home environments, semantic rules are used to reduce user involvement in data synchronization. An empirical study of the MemorySafe system shows that using characteristic-entry logs would reduce the amount of system resources such as memory, disk storage, and network bandwidth used for storing and exchanging logs to as little as 72% of the amount of system resources required when using normal logs. In the case of directories, configuration files and textual files, using characteristic-entry logs would reduce resource usage to 44%, 43%, and 33%, respectively. After that, several applications built on top of the MemorySafe system are described, which serve to validate the design of the MemorySafe system and to show how extra functionality can be added on top of the design of the MemorySafe system. One of them is "photo concept browsing", a new way of photo browsing, which is an application of Formal Concept Analysis. This application illustrates how formal theories can be applied in user-centric design in practice.

Samenvatting

Dankzij een snelle ontwikkeling van netwerk-technologie en voortschrijdende miniaturisatie, verschijnt er steeds meer consumentenelektronica in de huisomgeving in de vorm van onderling communicerende apparaten. Naarmate een verscheidenheid aan digitale opslagsystemen in de thuisomgeving terechtkomt, groeit de behoefte aan systemen die mensen in staat stellen zelf hun data op een eenvoudige wijze te beheren. Dit proefschrift beschijft het ontwerp van de MemorySafe, een gedistribueerd data management systeem voor thuisomgevingen. Het bewaren van de consistentie tussen meerdere kopieën van data objecten, die mogelijk gewijzigd worden op een moment dat de opslagapparaten niet met elkaar kunnen communiceren, is een probleem dat gebruikers van draagbare apparatuur veelvuldig tegenkomen (het zgn. 'disconnected updates' probleem). Eén van de drijfveren achter het ontwerp van de MemorySafe is het kunnen onderzoeken van datasynchronisatiemethoden voor thuisomgevingen.

In het eerste deel van dit proefschrift worden heterogeniteit, dynamiek en gebruiksvriendelijkheid geïdentificeerd als de kenmerken van thuisomgevingen die het ontwerp van data management systemen beïnvloeden. Daarna volgt een uitgebreid overzicht van bestaande systemen en gereedschappen die betrekking hebben op het disconnected updates probleem. De meeste systemen voor professioneel gebruik houden geen rekening met de kenmerken van thuisomgevingen, en zijn bovendien empirisch ontworpen zonder een zorgvuldige probleemanalyse. Vervolgens wordt een aantal gebruiksscenario's met betrekking tot disconnected updates gepresenteerd. Deze scenario's zijn gebruikt voor een uitgebreid onderzoek naar verschillende typen disconnected updates en de gevolgen daarvan, met behulp van de Z notatie. De conclusie is dat synchronisatie gebaseerd op identiteit en voorgeschiedenis van data objecten geschikt is voor data management systemen in huis.

In het tweede deel van dit proefschrift wordt een formeel model opgesteld in de Z notatie van een systeem dat disconnected updates ondersteunt. Eerst worden de specificaties en de consistentie-eisen formeel beschreven. Daarna worden standaard logs, die de voorgeschiedenis van updates bevatten, formeel beschreven en onderzocht. Vervolgens wordt een nieuw log-mechanisme geïntroduceerd, namelijk characteristic-entry logs. Een formeel model hiervan

wordt beschreven, en er wordt zorgvuldig bewezen dat characteristic-entry logs gebruikt kunnen worden in semantische datasynchronisatie op dezelfde manier als standaard logs. Een dergelijke formele behandeling van characteristic-entry logs rechtvaardigt het gebruik ervan voor datasynchronisatie in praktische toepassingen.

In het derde deel van dit proefschrift wordt het ontwerp en de implementatie van het MemorySafe systeem beschreven. Om met de heterogeniteit in thuisomgevingen om te gaan worden identiteiten van data geïntroduceerd, die model staan voor digitale multimedia data objecten welke vaak gekopieerd worden tussen apparaten onderling. Om de dynamiek van thuisomgevingen het hoofd te bieden, worden characteristic-entry logs ingezet om disconnected updates op te slaan in de MemorySafe, zodat minder systeem resources worden gebruikt en de datasynchronisatie sneller verloopt. Om gebruiksvriendelijkheid te garanderen zijn semantische regels gebruikt in het systeem, die de benodigde interventie van een gebruiker in het proces van datasynchronisatie verminderen. Een empirisch onderzoek naar het MemorySafe systeem laat zien dat characteristic-entry logs de belasting van geheugen, schijfruimte en netwerkbandbreedte tengevolge van opslag en uitwisseling van logs beperken tot slechts 72% ten opzichte van de systeembelasting van standaard logs. Voor directories, configuratiebestanden en tekstbestanden daalt dankzij characteristic-entry logs de systeembelasting tot respectievelijk 44%, 43%, en 33%. Tot slot wordt een aantal applicaties beschreven die gebruik maken van het MemorySafe systeem, welke dienen ter validatie van de MemorySafe en tevens als voorbeeld voor het toevoegen van extra functionaliteit aan het MemorySafe systeem. Eén applicatie is ‘photo concept browsing’, een nieuwe methode om foto’s te bekijken gebaseerd op Formal Concept Analysis. Deze applicatie laat zien hoe formele theorieën in de praktijk kunnen worden toegepast voor gebruiksvriendelijk systeemontwerp.

Curriculum Vitae

Yuechen Qian was born on 1 September 1974 at Wuxi, China. He moved with his parents to Nanjing when he was five. In 1992 he graduated from the Middle School Affiliated to Nanjing Normal University.

From 1992 to 1996 Yuechen studied computer science at the Department of Computer Science and Technology of Nanjing University. He obtained his B.Sc. degree in July 1996. After that, Yuechen studied formal semantics of parallel programming languages at the same department. He received his M.Sc. degree in July 1998.

After his graduate study, Yuechen followed the Software Technology program of the Stan Ackermans Institute at Eindhoven University of Technology, the Netherlands.

In July 1999 Yuechen joined the Eindhoven Embedded Systems Institute of Eindhoven University of Technology. Since then, he has worked as a Ph.D. student in the “Perceptive Home Environments” project at Philips Research Laboratories in Eindhoven.

Titles in the IPA Dissertation Series