Master Thesis

Flexible metadata-based partial synchronization for personal mobile devices

Author(s):
Alter, Yves

Publication Date:
2009

Permanent Link:
https://doi.org/10.3929/ethz-a-005899169

Rights / License:
In Copyright - Non-Commercial Use Permitted
Flexible metadata-based partial synchronization for personal mobile devices

Yves Alter

April 2009

Supervisors:
Dr. Oriana Riva
Prof. Gustavo Alonso

Systems Group
ETH Zurich, Department of Computer Science
Acknowledgments

Special thanks go to Dr. Oriana Riva for supervising this work and giving me many helpful hints. I would also like to thank Prof. Gustavo Alonso for his advice and giving me the opportunity to write this thesis in his group. A big thanks to my friends Ken Lee, Thomas Lenherr, Michael Gubser and Christoph Studer for the valuable linguistic and technical feedback they provided. Finally, I would like to thank my family for supporting my studies. You all rock!
Abstract

Personal data synchronization becomes more and more important as the number of devices a user possesses increases. Furthermore, data is now bigger in count and size and also much more versatile as ever before. Metadata is a possibility to describe such data in more detail. The main objective of this Master’s Thesis is to apply metadata to data synchronization that can be used with data and metadata of various sources. The result of the work is a framework that can be run on a variety of systems, from stationary desktops to handheld devices.
# Contents

1 Introduction
   1.1 Motivation .............................................. 1
   1.2 Related Work ........................................... 2
       1.2.1 Filesystem Synchronization ...................... 3
       1.2.2 Database Replication ............................ 3
       1.2.3 Distributed Storage System .................... 3
       1.2.4 Metadata Based Replication .................... 4
   1.3 Contribution ............................................ 4

2 System Overview
   2.1 Definitions ............................................. 6
   2.2 Sources of Data ....................................... 7
       2.2.1 Local Data ..................................... 7
       2.2.2 Remote Data ................................... 8
   2.3 Sources of Metadata .................................. 8
       2.3.1 Internal Metadata ............................... 8
       2.3.2 External Metadata ............................ 9
       2.3.3 Hybrid Metadata .............................. 9
   2.4 Subscriptions and Filters ............................ 9
       2.4.1 Types of Subscription .......................... 10
   2.5 Record Synchronization .............................. 10
       2.5.1 Logical Time in Distributed Systems ........... 10
       2.5.2 Capturing Causality between Events .......... 11
       2.5.3 Tracking Version of Records .................. 13
       2.5.4 Updating Vector Time Pairs ................... 16
       2.5.5 Synchronization Results ....................... 17
       2.5.6 Synchronization Protocol ....................... 18
   2.6 Consistency Models and Concurrency Control ....... 18

3 Node Architecture
   3.1 Definitions ............................................ 20
   3.2 Data Adaptation ....................................... 21
       3.2.1 Data Store .................................... 21
       3.2.2 Adapters ...................................... 22
       3.2.3 Outputs ...................................... 23
       3.2.4 Data Flow .................................... 24
       3.2.5 Examples ...................................... 24
   3.3 Communication Infrastructure ....................... 24
CONTENTS

Bibliography 60
A Data Stores I
B Adapters II
C RecordLock IV
Chapter 1

Introduction

Ever since computers started to exchange data through networks, data synchronization and replication has been an active area of research. In the past, mainly large data-centers replicated their data to other sites, either to increase availability, to distribute load or as a backup method, usually in some form of database replication. The network topology was well known and did not change over time, the network links were stable and so were the nodes. There was almost no demand in data exchange for personal use, since most users had only one system anyway. And if data was “synchronized” between systems it was usually done manually by copying it to a transportable media. The situation changed drastically when mobile devices became available at affordable prices and connectivity between devices started to be almost ubiquitous, be it once through infrared or cable and now through short range radios, wireless LAN, or cellular high-speed networks. Most individuals now have multiple devices that have the capability to store and process data. Today’s mobile phones, for example, have about the same processing power and storage that personal computers had a decade ago. The same applies for the recently introduced types of ultra-portable notebook computers which are descriptively called “netbooks”. Data synchronization between all these devices is nowadays a hot topic for many people and it is often solved similarly as it has been 15 years ago – by copying it to a USB stick or sending it by e-mail.

We continue this introduction by explaining the motivation, followed by related work and conclude the chapter by describing the contribution of this thesis.

1.1 Motivation

Modern systems are capable of handling a wide spectrum of data. People store documents, images, music and videos on their devices and often use some kind of application to manage each specific type of data. In the past, data was just identified by its filename on the filesystem. However, a hierarchical structure is often not flexible enough to describe or categorize specific data. Instead, metadata, data describing the data that is, can be used. Metadata can be an intrinsic property of the data that is defined by the content itself. For example, an image has a width and a height in pixels or an audio file has a length in...
seconds. On the other hand, metadata can also be defined extrinsically by the user. Such metadata is either not present in the file at all, not interpretable by a computer, or may also be a user's subjective perception. The location an image was taken at or the people or objects that are shown on it are one example, rating or cover art of a music track are another. It is of course possible to store extrinsic data again in the data-stream using some specific format, but it can also be stored externally, like in a database.

To distribute data among multiple devices it would be desirable to make use of metadata. This would allow to define the relevant set of data based on the metadata. For example, the user could decide to synchronize only music of a specific artist or only data that is part of a specific project. This type of data distribution is called partial replication. Another eligible property of a synchronization system would be to browse through the set of available (meta)data, even if the content is not available locally. A mobile music and video player with restricted storage could make use of such a feature by letting the user configure the system to replicate all metadata of type music and video to the device except for the actual content. He can then browse through his video collection directly on the device and fetch only the video he wants to watch. In this case, a data item is only partially synchronized.

If the user makes modifications to the data, they should be propagated to all devices that have a copy of that data. Changes should of course never be overwritten by other concurrent modifications, but in case there are no conflicting changes no user intervention should be required.

Devices usually connect and reconnect sporadically and synchronize data mutually between each other, but there is no specific order in which they connect. Consequently, data synchronization may occur in an order that forms (time multiplexed) loops. We therefore want a system that does not impose any restriction on the network connectivity and topology. Finally, devices should be kept in sync as long as they are connected, without an explicit sync operation.

In summary: a system for data synchronization to feature-rich, but potentially resource constrain devices should have the following properties:

- Use metadata to define the set of data to be replicated
- Allow partial replication
- Allow to replicate metadata only
- Work on any network topology
- Work on any connection behavior
- Keep nodes in sync as long as they are connected

1.2 Related Work

In this section, we will cover some examples of existing synchronization and replication technologies that are either well-established or still ongoing research.

---

1. In the sense of only parts of the whole set of the data.
2. In the sense of only parts of the data item, namely the metadata.
3. 1. A syncs with B; 2. B syncs with C; 3. C syncs with A
We discuss tools that synchronize files, replicate databases, systems that provide distributed storage and also metadata-based approaches.

1.2.1 Filesystem Synchronization

Tools that synchronize filesystems share the property that they replicate a subtree on the filesystem on one system to another. They may have different characteristics. Rsync, for example, is a one-way mirroring tool. The user specifies the direction in which the data should be sent and the tool will transfer only the changed parts of the data. Rsync does no real synchronization and therefore does not recognize conflicting changes, but it can at least skip files that have a newer time-stamp on the target device. Unison is similar to Rsync, but provides real synchronization and detects conflicts. It is designed for 1:1 synchronization, but also works with star-like topologies, where there is a central storage of data and the hosts synchronize pairwise with this storage. Though, if nodes are synchronized in cyclic order it is possible that updates are lost or deleted records reappear. Tra\textsuperscript{[1, 2]} or EnkelDS\textsuperscript{[15]} solve this problem by using vector time pairs as a basis for synchronization. The first one uses a custom sync protocol, whereas the second encapsulates the sync in a OMA DS\textsuperscript{[7]} protocol.

1.2.2 Database Replication

To increase availability and distribute load, a lot of database management systems support replication. Traditional eager replication systems that maintain consistency using two phase locking do not scale well and lazy replication can lead to conflicting transactions. Ganymed\textsuperscript{[10]} is a middleware that solves these problems by separating read-only transactions and write transactions. The first are handled by a set of slaves that provide snapshot isolation whereas the second are executed on a single master copy. Using snapshot isolation, the read-only transactions do not block update transactions while consistent reads are still guaranteed.

1.2.3 Distributed Storage System

A distributed filesystem makes distributed data available on the system as if it was stored locally. Usually, they are tied to the operating system transparently and the user accesses files in distributed filesystem by means provided by the operating system. An older, but well-known distributed filesystem is NFS, that was developed to make data on a system available to client systems in a transparent manner. It does not support client-side caching, which means that files are not accessible if the network link is down, but then it provides the same semantics as files that are stored locally.

Coda is more sophisticated, as it allows to operate even if it is not connected to the network, by caching data on the client. Even though Coda requires a server infrastructure, it is distributed and the data is automatically replicated between the different servers. Coda clients can not directly exchange updates between each other without the server infrastructure.

\textsuperscript{4}Formerly known as SyncML.
A system that is less a distributed filesystem than a distributed collaboration system is Bayou [14]. It was designed explicitly for offline operation by using weak consistency replication and detected and solved update conflicts using application-specific mechanisms. Consequently, applications must be aware of Bayou and have to use the provided libraries to access the data. On the other hand, Bayou has no support for partial replication.

PRACTI [3] is a replication system that provides partial replication, arbitrary consistency models and topology independence. The flexibility of PRACTI even allows to emulate existing replications systems by adjusting its parameters. Like Bayou, it uses a log-exchange protocol that reduces data transmission between nodes.

A very modern system is Wuala, that goes into the direction of cloud computing and even makes use of metadata. In Wuala, all nodes belong to a single big storage cloud and every node can choose how much local storage it wants to allocate to the cloud. Data that is shared using Wuala is then distributed to other nodes in redundant, encrypted chunks. This allows to reconstruct the data even if some nodes are disconnected from the network. Wuala can be used to either share data with other people as well as a backup mechanism and as central storage. It is possible to browse through the files in Wuala either in form of a network drive or through a dedicated user interface which also displays rich metadata about every file. On the other hand, it requires a central infrastructure run by the company develops Wuala and does not work without Internet connectivity.

1.2.4 Metadata Based Replication

Of course, there are also systems that focus on metadata, which do not just provide a filesystem semantic, but rather have an application centric approach. For example, the Roma Personal Metadata Service [13] builds a central store of metadata of a user’s data. This allows the user to keep track of different versions and locations of his data whereas the data itself is stored in a distributed manner among the users devices. Applications that want to make full use of Roma are required to be adjusted, which applies for their user interfaces as well as their I/O routines.

A new system that puts the devices into the center is PodBase [11]. It is a self-managed system, designed prevalently for mobile systems, that replicates data of each device to other devices to use them as redundant storage. Its primary goal is to ensure durability, which means if a node was to fail, the remaining nodes should be able to reconstruct the lost data.

Cimbiosys [12] is a novel system that extends WinFS [6], that is a folder-based partial replication system with content filtering which is very similar to what our proposed system does.

1.3 Contribution

We propose a synchronization framework that meets the criteria described in section 1.1. It is completely distributed and as such requires no central server instance. The system features a flexible data adaptation mechanism that allows...
building a custom adaptation scheme for various sorts of data. This is possible for different physical data locations as well as different logical composition of data and metadata. Data is accessed directly through the adaptation mechanism without requiring additional storage within the system. Items of data that are synchronized to other nodes are stored in the filesystem of the receiver where they are immediately available to other applications without additional adjustment. Data can be replicated partially, whereas the set of data is determined by specifying filters that are applied to metadata of content. Those filters can be subscribed at other nodes, which then continuously inform nodes about changes to the affected data as long as they are connected.

The framework is a user-level application written in Python and can be run on almost all platforms that provide a socket interface.
Chapter 2

System Overview

The proposed system consists of two main parts. One part is an interface that allows to adapt a wide range of data sources using a flexible adapter mechanism. The other part is a communication infrastructure that features a data synchronization protocol to distribute the adapted data based on its metadata.

In this chapter, we will give an overview of the whole system, describe different forms of data and metadata, give an introduction to the concept of logical time as used in distributed systems and explain the synchronization protocol.

Figure 2.1 is a top-view at the whole system. It consists of multiple devices that can mutually exchange data through two different channels, one for command and metadata exchange and another for content transfer. A node is the software component that runs on every device and implements the actual system. Every node has two sources of data attached: local data, that is already present in some sort of storage on the device, and remote data, that was received from another node. The user of the device can access remote data by just accessing the filesystem and local data depending on the type of data and storage technology.

2.1 Definitions

Before we introduce the proposed system in detail, we define some terms used in this chapter.

**System** A set of nodes that exchange data.

**Device** A computing device that contains storage capabilities. This can also be a resource-constrained device, like a smart phone or an Internet tablet.

**Node** A software module on a device that provides APIs and storage attachment facilities. Nodes have the ability to discover and communicate with other nodes.

**Data** The actual resource the user is interested in. Typical examples are text documents, images, audio tracks and videos.

**Content** A stream of bytes which represents the raw part of the data.

**Record** A specific entity of data, like a file or a database row.
CHAPTER 2. SYSTEM OVERVIEW

Metadata Describes either the resource itself, for example the size of a file, or the content of the resource like the title of a music track.

Data type The high level type of the data like document, image, video, etc.

2.2 Sources of Data

We already described other systems that allow to use different mechanisms to specify and retrieve content of interest from a specified data source. The class of filesystem synchronization tools, for example, that replicate parts of a directory tree to another (usually remote) filesystem operate by definition on files only. The situation is similar for database replication mechanisms, where the replica is again a database. Distributed storage systems require the user to move their data into the system to make it available, which can either be realized transparently through operating mechanisms or using a dedicated user interface. Our proposed system takes another approach by differentiating between local data and remote data.

2.2.1 Local Data

Data stored on a device, that is meant to be made available to other nodes is called local data. We do not explicitly define the kind of data storage local data is backed by, but rather we want to provide flexible and uniform ways to access it. We will later see how the idea of source adaptation will provide us such an abstraction. For the moment it is sufficient to note that data that belongs to the set of local data is under full control of its owner and can be modified in arbitrary ways, independent of our proposed system.
CHAPTER 2. SYSTEM OVERVIEW

<table>
<thead>
<tr>
<th>Data type</th>
<th>File format extensions</th>
<th>metadata fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document</td>
<td>.odf, .doc, .pdf</td>
<td>author, summary</td>
</tr>
<tr>
<td>Image</td>
<td>.jpeg, .png, .tiff</td>
<td>dimensions, shot date, exposure time, geolocation</td>
</tr>
<tr>
<td>Audio</td>
<td>.mp3, .aac, .ogg</td>
<td>play time, artist, title, album</td>
</tr>
</tbody>
</table>

Table 2.1: Examples of metadata stored as part of a file

2.2.2 Remote Data

The counterpart of local data is remote data, which is data received from other nodes. There are multiple ways how to allow the user to access this data on a node. One way is to store data in a system's internal storage, for example a database, and expose it through an API of the system. The advantage is that the system has full control over the data and it may also be augmented with metadata not directly available in the content itself. The evident disadvantage is that the user is not able to access the data through ordinary operating system interfaces only. This may be a significant problem in a scenario, where users want to use third party applications with remote data, that cannot be adjusted to work with another storage technology. We decided to take a hybrid approach, in such a way, that data received from other nodes is stored in the local filesystem, directly accessible by the user without any restrictions. Additionally, nodes provide an API allowing extended operations like modifying and browsing through metadata and committing changed content back to the system.

2.3 Sources of Metadata

Metadata enriches the actual content with additional information which allows a more precise organization of data than it is possible with hierarchical storage like filesystems, for example. Since technologies like WinFS, that integrate metadata directly into the storage system have yet to become available in productive systems, the question that arises is where metadata is stored. We can distinguish between the following three possibilities that we will describe based on some examples.

2.3.1 Internal Metadata

Metadata can be stored directly in the actual data format. This can be in the specification of the file format as well as in a dedicated container format, which combines raw data and metadata streams into a single byte stream.

Example Audio file container formats like MPEG-4 Part 14 or MPEG-1 Audio Layer 3 (MP3) support dedicated fields for information like artist, album, title, etc. Similarly, the exchangeable image file format (EXIF) specifies how metadata tags for images are integrated in file formats like JPEG or TIFF. Table 2.1 provides some examples for internal metadata.
2.3.2 External Metadata

External metadata is stored and usually also maintained totally separately from the actual content. This is often the case for multimedia data that is typically managed through specialized applications. In most cases, they store metadata, either supplied by the user or derived from the content itself, in some sort of data store separate of the actual data.

Example The Banshee Media Player has the ability to import audio files into a library and stores extra metadata like rating or play count in a dedicated SQLite database. Existing (internal) metadata is extracted during the import and also stored in the same database. A special case is album cover art, which is saved as a file in the filesystem, in the directory where the files that belong to the album are located. Therefore, Banshee is an example that interfaces with different sorts of external metadata.\footnote{For the sake of completeness, note that Banshee optionally supports writing changed metadata back into the file.}

2.3.3 Hybrid Metadata

It is possible to have any combination of internal an external metadata, of course:

Example Imagine an image management tool like Picasa or iPhoto, which usually keeps metadata already present inside the file, but maintains extended information like descriptions, ratings or album membership of images, externally.

Because we want to specify the set of data a node is interested in based on metadata, we will maintain metadata as part of the node itself and provide the user with an API. The details of how to extract metadata from data will be covered in detail in section 3.2. For now, we suppose we already have the data available at the node.

2.4 Subscriptions and Filters

To define the set of data a node is interested in, we decided to introduce data filters. They consist of Boolean compositions of filter elements and other filters, making it possible to nest them. Filter elements describe comparisons between metadata and their designated values. Since a filter operates on metadata, it always has an associated type and therefore only applies to (meta-)data of that type. This set differs for every node and by using filters it is possible to only synchronize the set of all data partially.

Nodes subscribe their filters to other nodes, so that they can be notified if data is updated (see section 2.5). As long as nodes are connected, changes are always propagated without requesting an explicit sync.

\footnote{For the sake of completeness, note that Banshee optionally supports writing changed metadata back into the file.}
2.4.1 Types of Subscription

Our system distinguishes between metadata\(^1\) and content and transfers them over two distinct channels (see figure 2.1 and section 3.3). This allows us to support two different types of subscriptions called partial and full. A partially subscribed filter means that the node is only interested in the state and the metadata of records that match this filter, but not in the actual content. A full subscription consequently expresses the fact that a node is also interested in the content. In such a system, changes to metadata and content do not necessarily occur at the same time.

Changes to data can either occur if data was changed locally or during a sync with another node. In both cases, at least the metadata has changed and notifications are sent to all nodes that have at least one matching subscription, partial as well as full. Possibly, the content will be changed as well and nodes that match at least one full subscription will receive an additional content notification.

Such a design has several advantages:

- A node resides in the state of synchronization for a shorter time, because less data is transmitted.

- Synchronization decisions can be taken without knowing (or storing) the whole content.

- Nodes are able to follow the state of a set of data without requiring to store the whole content on the device.

- Nodes are able to determine freely when and from where they eventually want to fetch the actual content.

This makes it possible, for example, to browse through the data available in the system, select a record of interest based on its metadata and fetch the content for that specific record only.

Data that have no subscription will be completely ignored by the node and require neither storage on the device nor bandwidth during synchronization.

2.5 Record Synchronization

We have already seen that data on a node is either local, attached through sources, or remote, received from other nodes using subscriptions. This section starts with an introduction to logical time in distributed systems, which is required to understand how data and relations between different versions of data are tracked and used during synchronization. Afterwards, we talk about when synchronization messages are exchanged and we will conclude with a discussion of modification to concurrency control.

2.5.1 Logical Time in Distributed Systems

To track the order of events in a distributed system it is usually not suitable to use a global time, because it might not exist or it is hard to keep it synchronized\(^2\) In this context, metadata is both maintenance and content metadata.
between nodes. Instead, a logical time can be introduced that let us order events chronologically and causally based on a happened-before relation. An event $a$ is said to have happened-before an event $b$ ($a \rightarrow b$) iff

- $a$ occurs before $b$ on the same process,
- $a$ is a send event on one node and $b$ is a receive event of the same message on another node or
- there is an event $c$, such that $a \rightarrow c$ and $c \rightarrow b$ (transitive closure).

Figure 2.2 illustrates the happened-before relation on a process-time diagram. The following are some examples that result if we apply the rules we just defined:

- $a_1 \rightarrow a_4$, $b_2 \rightarrow b_3$, $c_1 \rightarrow c_2$
- $a_1 \rightarrow b_1$, $b_2 \rightarrow a_3$, $b_4 \rightarrow c_2$
- $a_1 \rightarrow b_2$, $c_1 \rightarrow c_2$, $a_1 \rightarrow c_2$

As we can see, $a \rightarrow b$ is satisfied if there exists a path from $a$ to $b$ in the process-time diagram.

### 2.5.2 Capturing Causality between Events

To computationally decide whether events are causally related or not, we need to define a clock and assign each event an element of its domain. There are multiple protocols that implement such a clock, each with different capabilities. As introduction, we will shortly discuss two of them.

**Lamport clock**

The Lamport clock is a scalar value that is assigned to each event. It is implemented by storing a counter at every node in the system. Before each event, the (local) counter is incremented and the new value is assigned to the event. In case of a send event, the incremented value is attached to the message and sent to the other node. For a receive event, the counter is set to the maximum $^3$or “virtual”.
CHAPTER 2. SYSTEM OVERVIEW

(a) Causally related events

Figure 2.3: Lamport clocks do not capture concurrent events

of the local counter and the clock value that was received in the message and will then be incremented. If we denote \( C(e) \) as the time-stamp assigned by the Lamport clock protocol for an event \( e \), the following holds:

\[ a \rightarrow b \Rightarrow C(a) < C(b) \]

Therefore, if an event \( a \) is the cause of an event \( b \) (they are causally related), the time-stamp of \( a \) must be less than the time-stamp of \( b \). If we apply the contraposition we can conclude that

\[ C(a) \neq C(b) \Rightarrow a \not\rightarrow b \text{ or } C(a) \geq C(b) \Rightarrow a \not\rightarrow b \]

The last equation holds because of the definition of the less-than relation on natural numbers, and allows us decide if an event is not causally related to the other.

The problem with Lamport time-stamps is that their ordering is only a necessary but not a sufficient condition for causally related events, which means we can not distinguish between causally related or concurrent events based on the time-stamps:

\[ C(a) < C(b) \Rightarrow a \rightarrow b \]

\[ C(a) < C(b) \Rightarrow a \rightarrow b \vee a \parallel b \]

If an event \( a \) has a smaller time-stamp than another event \( b \), we can not conclude that they are causally related, because they may have happened concurrently (\( a \parallel b \)). This is obvious and we can easily identify a counter-example in the process-time diagram as proof: figure 2.3a illustrates three events that are causally related and subsequently fulfill the Lamport criterion. In figure 2.3b another event \( a_2 \) happens on node A and gets a time-stamp of 2 assigned. Even though it is lower than the time-stamp of \( b_2 \), they are not causally related but happened concurrently.

4of the receiving node
CHAPTER 2. SYSTEM OVERVIEW

Vector Clock
To decide if two events are either causally related or concurrent, we need to extend the Lamport clock to a mechanism called vector clock. Instead of storing a scalar value, every node tracks the counter of all nodes in the system in a vector. At each event, only the component in the vector with the local counter is incremented. At a receive event, the local clock is set to the component-wise maximum of the local clock and the received vector.

Time-stamps of Lamport clocks could be compared naturally, because they are integers, but to compare time-stamps of vector clocks we need to define a partially ordered set. If we denote $VC(e)$ as the time-stamp assigned to an event $e$ using the vector clock protocol, we define the non-strict partial order $\leq$ and its strict form $<$ as follows:

$$VC(a) \leq VC(b) \Leftrightarrow \forall x [VC(a)_x \leq VC(b)_x]$$

$$VC(a) < VC(b) \Leftrightarrow VC(a) \leq VC(b) \land \exists x [VC(a)_x < VC(b)_x]$$

We say, the time-stamp of an event $a$ is less than the time-stamp of an event $b$, if every element in $a$ is less or equal to the corresponding element in $b$, whereas at least one element must be strictly less. Note that for the moment we assume that for every node in the system there exists an element in the vector. We will later relax this restriction and consider version vectors as partial functions.

In contrast to Lamport clocks, the time-stamps of vector clocks can be used to determine if two events are causally related:

$$a \rightarrow b \Leftrightarrow VC(a) < VC(b) \tag{2.1}$$

An event $a$ is said to happen concurrently ("at the same time") as an event $b$ if neither one happened before the other:

$$a \parallel b \Leftrightarrow \neg (a \rightarrow b) \land \neg (b \rightarrow a) \tag{2.2}$$

If we substitute (2.1) into (2.2) we can computationally decide if two events are concurrent based on vector clock time-stamps:

$$a \parallel b \Leftrightarrow \neg (VC(a) < VC(b)) \land \neg (VC(b) < VC(a))$$

2.5.3 Tracking Version of Records
We now have a mechanism to order events in a distributed system and decide if they are causally related or not. After having a look at how to apply logical clocks as version vectors as a tool to synchronize different versions of data, we will see why they are not sufficient and describe an extended mechanism, the vector time pairs.

5 We assume every node has already a Universally Unique Identifier (UUID) assigned.
Version Vectors

Vector clocks can be used to track versioning of records. We assign to each record a vector clock that tracks the time of its last modification. In contrast to the vector clock algorithm just described, nodes only need to maintain a scalar counter. It is incremented at each event, similar to Lamport clocks. A modification to a record is captured by setting the local component of the record’s vector clock to the current value of the counter of the node.

By comparing the time-stamp of different versions of a record makes it possible for the synchronization algorithm to decide which action to take. If two versions of a record are being synchronized, one version can be causally dependent on the other or the modification happened concurrently. For simplicity, we denote \( \alpha, \beta, \ldots \) as versions of a specific record \( r \), instead of using the complete form \( VC(r_\alpha), VC(r_\beta), \ldots \), and get:

\[
\begin{align*}
\alpha &= \beta : \alpha \text{ and } \beta \text{ are identical} \\
\alpha &< \beta : \beta \text{ is a successor of } \alpha \\
\beta &< \alpha : \alpha \text{ is a successor of } \beta \\
\alpha \parallel \beta : \alpha \text{ and } \beta \text{ are conflicting}
\end{align*}
\]

The action of a one-way synchronization of a record from node A to node B (\( A \xrightarrow{} B \)) can now be decided by comparing the version vectors:

\[
sync_{A \xrightarrow{} B}(\alpha, \beta) = \begin{cases} 
\text{skip} & \alpha = \beta \vee \alpha < \beta \\
\text{replace } \beta \text{ with } \alpha & \beta < \alpha \\
\text{conflict} & \alpha \parallel \beta
\end{cases}
\]

We can illustrate these cases in a process-time diagram with two nodes and a single one-way synchronization of a record, as shown in figure 2.4.

Figures 2.4a to 2.4c require no further consideration, because the sync either leads to no action (2.4a and 2.4b) or replaces the record (and thus its version) with its successor (2.4c). A conflict (2.4d) needs to be resolved by one way or another and the resolution should be recorded, of course. There are two possibilities to resolve a conflict:

- decide for one of the existing versions or
- create a new version by merging the conflicting records.

In either case, the resulting version is a successor of the conflicting records and that fact should be reflected by the version vector. Therefore, the question comes up of how to update the version vector after a conflict resolution. A version vector for a merge can be built by taking the element-wise maximum and incrementing the counter for the local element. This is equivalent to a receive operation described above and will capture that both conflicting versions happened before the new version, which is subsequently a successor. If the resolution was to use an existing record, we could either keep the version vector

---

\(^6\)A modification is therefore a concrete instance of the generic events previously discussed.

\(^7\)A two-way synchronization can be obtained by two subsequent one-way syncs in each direction.
as it was or update it as if a normal modification happened. As it turns out, neither is a solution: If we choose to keep the version vector, we would lose the fact, that the conflict has been resolved at all. And in the second case other nodes would detect that their version is outdated and would decide to replace it with the supposed new one, even though it might be identical to the resolved one.

We can conclude that version vector are not sufficient to store both modification histories and conflict resolution decisions.

**Vector Time Pairs**

A solution are vector time pairs, as proposed in [1, 2]. Instead of using a single vector that tracks modifications only, vector time pairs are two distinct vectors: a modification vector \( \vec{m} \) and a synchronization vector \( \vec{s} \). The first one behaves like a version vector described in section 2.5.3, whereas the second one captures modification and synchronization events together, so \( \vec{m} \) is always contained within \( \vec{s} \). The idea is now to decide about the synchronization action by comparing \( \vec{m} \) and \( \vec{s} \) of two different versions of a record.

As already mentioned, we will now consider \( \vec{m} \) and \( \vec{s} \) as partial functions that are subsets of \( UUID \times N \), that map a node identifier to a time-stamp. Again, we first define a partial order between those partial functions:

\[
\vec{m} \leq \vec{s} \iff \text{dom}(\vec{m}) \subseteq \text{dom}(\vec{s}) \land \forall x \in \text{dom}(\vec{m}) : \vec{m}(x) \leq \vec{s}(x)
\]

\( \text{dom}(\vec{m}) \) and \( \text{dom}(\vec{s}) \) are the domains of the partial functions (i.e. the UUIDs that have a time-stamp assigned). In other words, we say that \( \vec{m} \) is included in \( \vec{s} \), if and only if the domain of \( \vec{m} \) is a subset of the domain of \( \vec{s} \) and every

\[^8\text{Since the set of UUIDs is a (finite) countable set, we could also use the Cartesian product } N \times N.\]
CHAPTER 2. SYSTEM OVERVIEW

**Synchronization Decision**

\[ \vec{m}_{\text{new}} \]

\[ \vec{s}_{\text{new}} \]

\[ \vec{m}_\alpha \]

\[ \vec{m}_\beta \]

\[ \max(\vec{s}_\alpha, \vec{s}_\beta) \]

\[ \max(\vec{s}_\alpha, \vec{s}_\beta) \]

\[ \max(\vec{s}_\alpha, \vec{s}_\beta) \]

<table>
<thead>
<tr>
<th>vector</th>
<th>skip</th>
<th>replace</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>\vec{m}_{\text{new}}</td>
<td>\vec{m}_\alpha</td>
<td>\vec{m}_\beta</td>
<td>-</td>
</tr>
<tr>
<td>\vec{s}_{\text{new}}</td>
<td>\max(\vec{s}<em>\alpha, \vec{s}</em>\beta)</td>
<td>\max(\vec{s}<em>\alpha, \vec{s}</em>\beta)</td>
<td>\max(\vec{s}<em>\alpha, \vec{s}</em>\beta)</td>
</tr>
</tbody>
</table>

Table 2.2: Vector time pairs update rules

element in \( \vec{m} \) has a value less or equal to the corresponding value in \( \vec{s} \). This definition makes it possible to compare version vectors that do not “know” the same set of nodes in the system.

The synchronization action can now be decided by comparing the modification and synchronization vectors. If \( \vec{m}_\alpha, \vec{s}_\alpha \) and \( \vec{m}_\beta, \vec{s}_\beta \) denote the modification and synchronization vectors of a record on node A and B respectively, we can formulate the synchronization decision for a sync from A to B as follows:

\[
\text{sync}_{A \rightarrow B}(\vec{m}_\alpha, \vec{s}_\alpha, \vec{m}_\beta, \vec{s}_\beta) = \begin{cases} 
\text{skip} & \vec{m}_\alpha \leq \vec{s}_\beta \\
\text{replace } \beta \text{ with } \alpha & \vec{m}_\beta \leq \vec{s}_\alpha \\
\text{conflict} & \text{else}
\end{cases}
\] (2.3)

The synchronization vector can be thought of the state that we know of a record and the modification vector is the actual version of the record. This lets us describe the synchronization decisions naturally: If the version of the remote record \( \vec{m}_\alpha \) is older than the state \( \vec{s}_\beta \) we already know, we can skip the record. On the other hand, if our version is older than the remote state, we replace our copy with the remote one. If neither applies, a conflict occurred that needs to be resolved.

### 2.5.4 Updating Vector Time Pairs

After the synchronization action was decided, it is required to update \( \vec{m} \) and \( \vec{s} \). In case of a skip, \( \vec{m} \) will not be changed, and in case of a replace \( \vec{m} \) is set to the \( \vec{m} \) of the record that replaced the current one. In either case, the synchronization vector will be set to the component-wise maximum of both records. See table 2.2 for an overview of the update rules.

Version vectors suffered from the problem that they cannot record the resolution of a conflict. With vector time pairs, we have the possibility to encode conflict resolutions by updating the modification and synchronization vectors accordingly. Table 2.3 summarizes the update rules for a conflict during a one-way sync \( A \rightarrow B \). The new value of \( \vec{m} \) is either the one of the chosen copy, or in case of a merge it is treated as if a new version was created and the local counter of B is taken, which is a relation between the UUID of the node and precisely the counters value (\( (ID(B), C(B)) \)). To reflect the fact that the resolved record is a successor of both versions, the synchronization vector is set to the element-wise maximum of each \( \vec{s} \):

\[
\max(\vec{s}_\alpha, \vec{s}_\beta) = \{(x, y) | (x, y) \in (\vec{s}_\alpha \cup \vec{s}_\beta) \land \neg \exists z : (z > y \land (x, z) \in (\vec{s}_\alpha \cup \vec{s}_\beta))\}
\]

In case of a merge, we update the value of the local component to the current counter of the node. This can be achieved by taking the new \( \vec{m} \) into the maximum.
Besides the two vectors we have already covered ($\vec{m}, \vec{s}$), there is a third vector that stores the time of creation of every record ($\vec{c}$). The content of this vector is never changed once it has been created. It contains a single relation between the creation node and the time only. The creation vector also uniquely identifies a record, because only a single record can be created on a node at a specific point in time.  

### 2.5.5 Synchronization Results

So far, we dealt with tracking versions of records and comparisons between them. Based on this knowledge, we can derive the high-level results that may occur during a one-way synchronization of a record from $A \rightarrow B$.

**Adding Records**

A yet unknown record from $A$ is added to $B$.

**Deleting Records**

If the deleted record on $A$ is successor the (still existing) record on $B$, the result is a deletion. Deleted records can not be removed completely from the system. If we did so as long at least one node still has a copy of the affected record, it would not be possible to decide during a synchronization, if the record was deleted or just unknown and it would reappear. Instead, all associated data and metadata will be deleted and the record is flagged as deleted by removing all data from the meta record except the create and synchronization vectors. The create vector is required since it is used as an identifier and the synchronization vector allows to keep track of the state of the deletion.

**Replace Records**

If the record on $A$ is a successor of the record on $B$, a modification occurred. The old version will be replaced by the new one. In equation 2.3 we saw that replacements are detected by comparing the synchronization vector of $A$ against the modification vector of $B$. In case the record has been deleted on $B$, the modification vector is not available anymore. Instead, its synchronization vector will be used.

**Modify-Modify Conflicts**

A modify-modify conflict occurs if a record is changed on two nodes concurrently. In this case, neither record is a successor of the other and the conflict has to

---

Table 2.3: Conflict resolution encoding through vector time pairs

<table>
<thead>
<tr>
<th>vector</th>
<th>$\alpha$</th>
<th>$\beta$</th>
<th>merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m_{\text{new}}$</td>
<td>$m_\alpha$</td>
<td>$m_\beta$</td>
<td>${(ID(B), C(B))}$</td>
</tr>
<tr>
<td>$s_{\text{new}}$</td>
<td>$\max(s_\alpha, s_\beta)$</td>
<td>$\max(s_\alpha, s_\beta)$</td>
<td>$\max(s_\alpha, s_\beta, m_{\text{new}})$</td>
</tr>
</tbody>
</table>

---

9Time is again the local counter on the node
be resolved by either choosing a specific version or creating a new version (a merge) that is a successor of both.

Modify-Delete Conflicts
If a deletion happened concurrently to a modification, a modify-delete conflict occurs. Such a conflict has to be resolved by the user.

Delete-Delete Conflicts
Deletions are almost identical to ordinary modifications. There is a difference if two nodes concurrently delete a record, though: any possible resolution leads to a deleted record. For this reason, a delete-delete conflict can be resolved automatically.

Skip
In all other cases the result of the sync is \textit{skip} (no action).

2.5.6 Synchronization Protocol
Synchronization is implemented by a simple stateless protocol. As soon as a node connects to another the following two steps happen:

1. Subscription of filters
2. Request changed records

To subscribe filters, a node sends all its filters that define the set of record it is interested in to the other node. Subscriptions are stored persistently and are valid until a node recalls or modifies its subscriptions. This allows the other node to decide which nodes want to be notified in case a record or content has changed by applying the filters to it.

If a node reconnects, it is of course possible that it missed changes that happened during the time it was disconnected. To catch up with those changes, the node initiates a sync. It does this by sending a list of record IDs together with their synchronization vector. We saw in equation \ref{equation:2.3} that the only case where no further action is required is \textit{skip}. The remote node is therefore able to decide that the decision is \textit{not} skip based on just the synchronization vector of the node that initiated the sync. It then sends notifications (containing the whole meta records) for all records that have at least one matching filter (either partial or full) and whose synchronization decision is not \textit{skip}.

Content is not exchanged during this initial sync. Instead, every node determines itself if its content of a record is outdated and requests it through the second channel from another node that is available. Details about this are covered in section 3.4.5.

2.6 Consistency Models and Concurrency Control
Distributed systems can support different models of consistency. Traditional models are for example \textit{strict consistency} which is the strongest consistency
model and requires that all write are seen in the order they were executed. This would require a global clock for all nodes in the system and is therefore not possible anyway. A weaker and realizable version is sequential consistency, which requires that all write are seen in the same order by all nodes. But as soon as we want to support partial replicas and frequently disconnected nodes, such models are not suitable, since they actually impose a global state. Using the vector time pair protocol, we enable optimistic and partial replication of specific records. Optimistic, because the records may diverge and we assume that at some point the system regains a consistent state (when all nodes are connected and all conflicts resolved) and partial, because it is possible to only replicate a subset of data by just ignoring the data that are not part of a filter. As an alternative, pessimistic replication is also supported where only a single node may accept writes. It is important to note that we do not support consistency between records: every record is treated independently and are never causally related.

In our system we differentiate between local data, which is attached through sources, and remote data, which is received from other nodes. Local data can, by definition, be modified at any point in time and the system has no control over that. We say that records originating from local data are owned by the node that created it. If such records are sent to other nodes, they become remote data for the receiver. The system currently allows two possibilities to control concurrency of remote data. The first one is to allow writes to records from any node at any time. In this case, records diverge between nodes, but they will be synchronized again and the system will eventually get into a consistent state. Of course, conflicts can occur if a single record is modified concurrently and they have to be resolved manually by the user. The second possibility is a locking based approach where data can only be written if a lock from the creating node has been acquired before. The creating node then acts as a coordinator. This way, no conflicts can occur and the creating node can control who is allowed to commit modifications. On the other hand, no writes are possible if the owner of the record is not reachable and commits are much slower.

Whatever protocol is used to coordinate concurrent access to remote records, there is always the possibility that changes to local data lead to a conflict between the new version of the data source and the version received from another node. To overcome this problem, sources can be configured to be either soft or strong. Updates to records from strong sources will always trigger a refresh cycle first and conflicts can therefore be handled using the same resolution mechanisms as for remote data. Records from soft sources do not trigger this update and are immediately written.
Chapter 3

Node Architecture

So far, we covered the components of the global system and how the nodes in the system work together. In this chapter, the focus is on a single node. We will present its architecture and discuss selected components in detail.

Again, we start with a few definitions, followed by the introduction of the mechanism of data adaptation. In chapter 2, we assumed that other nodes are just known to each other and are able to communicate, so it is necessary to catch up with the communication infrastructure. Finally, purpose and function of some subsystems are explained.

Figure 3.1 is an overview of the whole architecture and serves as a reference point through this chapter.

![Node Architecture Diagram](image)

The Node layer is the central point of the system. It uses the Sync Manager to initiate and control synchronization operations, communicates with other nodes using the Remote Node interface, registers handlers for incoming requests at the Remote Node Server and stores its settings persistently with the help of the Persistent Property Manager.

### 3.1 Definitions

Before we discuss the components in detail, we define again some terms that are used throughout this chapter.

**Node**  Beside the definition given in chapter 2, a node is now also a component in the node architecture that glues the other components together and
provides an API to the user of the system. The actual meaning should be obvious out of the surrounding context.

**Peer** Another node in the system.

**Remote node** An object providing access to a remote node. The difference between peer and remote node is subtle, but whereas peer can be considered as a remote entity, a remote node actually provides a high level interface for this entity. In other words, a remote node is the local incarnation of a peer.

**Meta record** A data structure for a specific data record that contains both the maintenance data used by the system (for example the version vectors) as well as a type record.

**Type record** A data structure that contains the MIME-type of the data it describes as well as the metadata.

**Content record** A data structure that contains maintenance data about content available at the node.

### 3.2 Data Adaptation

In sections 2.2 and 2.3 we presented the problem of different, possibly unknown sources of data and metadata. If we intend to build a system that supports different sorts of data storage as well as extraction of associated metadata (either internal or external), it needs to have a mechanism that lets us define data sources in an abstract way. We propose the concept of data sources as a possible solution, which will allow both to accept different storage as inputs (section 2.2) and to handle different kinds of metadata (section 2.3). In the architecture schematics (figure 3.1), data adaptation covers the layers source and data store.

Figure 3.2 shows the components of a source. Every source is identified by a name and is responsible for a specific sort of local data (section 2.2.1). It transforms data to a set of outputs through the use of an adapter graph.

#### 3.2.1 Data Store

To get access to the local data, a source contains a single **DataStore**, which is a component that implements the interface to a specific physical store on a device, for example a sub-tree in the filesystem or a database. Data stores have the following functionality:

- Retrieve the identifiers for all records available
- Get the record for a specific identifier
- Change or delete a record

Obviously, data stores provide a stateless interface to a collection of records, but since the adapter graph only transforms a single input into an output (and can therefore be considered as a function free of side-effects), the gap is bridged by a **DataStoreAdapter**. It allows to select a specific record as active and makes its content available on an adapter-like interface.
Data stores can be configured by various options depending on the type of data store. For example, the *FilesystemDataStore* has options to configure

- a path on the file system,
- the possibility to scan directories recursively and
- to filter results based on regular expressions.

### 3.2.2 Adapters

Adapters take a set of inputs and adapt them to a set of outputs. An element of such a set is called *pin* and every output pin of an adapter can serve as input for another adapter by specifying a dependency between them. Our system comes with a predefined collection of adapters that fulfill various tasks. For example, the *FileStatAdapter* adapts a filename to outputs like creation time, modification time, file size, etc. Another important example is the *ContentMetadataAdapter*, which adapts a file-like object to the metadata that is stored within it, thus it adapts internal metadata.

Some adapters have associated *options*, that allow to configure the behavior of the adapter. Options are statically named properties of an adapter instance, which means they do not change over time, but it is possible to use multiple

---

1This does not mean that every combination is also valid. For example, it is required that the types of the input and output pins match.

2A file-like object is a stream of bytes that can be accessed through an interface that is similar to the one for ordinary files. See section 4.1.2 for a detailed description.
instances of the same adapter, each with different configurations. The ConstAdapter is an example that makes use of options: it emits the “value”-option on its single output pin.

There are two classes of adapters: read-only adapters and read-write adapters. Adapters that belong to the first class can only adapt data from input to output pins. The second class contains adapters that are invertible, that means they can provide an input given an output, similar to a bijective function. This becomes important, when we discuss the reintegretion of changed data back to the local data store in section 3.2.4.

For an extensive description of all adapters and their properties see appendix B.

3.2.3 Outputs

Sources have four predefined outputs (table 3.1). Depending on the data to adapt, the user of the system populates the source with different adapters, and specifies the flow of data between each of them, represented by the dashed arrows in figure 3.2.

A source is always associated with a specific high level type, for example “document” or “image”. Every type specifies a set of metadata fields whose values are made available as an output of the source called type record. It contains the metadata that was extracted from the data record by traveling through the adapter graph.

An anchor is an arbitrary value that is used by the system to determine if a data record was changed or not. For example, a hash of the content could be used as anchor, and the system would detect any change to the content as modification with the costs of more processing power required. Another possibility for an anchor value could be the modification time-stamp for data stores backed by a filesystem, which would be faster, but less accurate in terms of change recognition.

The last two outputs deal with the actual content that is the data contained in the data record. The system requires the values content size and file object, whereas the first is the size in bytes of the data and the second is a file-like object that can be used to access the data.

\[\text{Table 3.1: Source outputs}\]

<table>
<thead>
<tr>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Record</td>
<td>Contains metadata of the adapted record. Matches the type of the source.</td>
</tr>
<tr>
<td>Anchor</td>
<td>A value that is used to determine if the record contains modifications.</td>
</tr>
<tr>
<td>Content Size</td>
<td>Size in bytes of the content data.</td>
</tr>
<tr>
<td>File Object</td>
<td>A file-like object used to access the content data.</td>
</tr>
</tbody>
</table>

3\(^{3}\) The const adapter is special, because it does not have an input. Even though it is technically identical to an adapter, it behaves more like an emitter.

4\(^{4}\) Except for cases of hash collisions.

\[23\]
3.2.4 Data Flow

The adapter graph specified by the user describes a dependency graph, that defines the flow of data from a source to multiple sinks. The data flow from the data store to the outputs is straightforward and we will postpone the discussion of the responsible algorithms of adaptation to section 5.4. Besides data extraction (which happens by moving data in direction of the dependencies), some changes at the side of the output must also be propagated back to the data store, which is the process of integration. We can determine the set of output elements that are eligible for this process as follows:

1. Anchor and content size outputs are never written back.
2. Fields of a type record that are mapped directly to a data store output will be written back.
3. Fields of a type record that have a path originating from the data store that passes through read-write adapters only will be written back.
4. The file object output is written back if it has a path that originates either from a read-write adapter or from the data store and only passes read-write adapters.

Point 1 follows from the fact that anchors are an auxiliary construct to detect changes and are therefore read-only values by definition. The content size does not require to be updated either, because it will reflect a change of content implicitly (through re-processing the graph after an update). The second point is evident, because there is a one-to-one correspondence between an output pin of the data store and a type record field. Fields that are connected by intermediate adapters need to be stepwise integrated back to the data source and this is only possible if the property 3 is fulfilled. Note that this property does not guarantee that all output pins of a read-write adapter, which are required to integrate data to inputs are connected. The case is a little bit different for the file object output. It is important to understand that the content represented by the file object, is not necessarily stored at the data store. Instead, it may also originate from an adapter, which means that it is not always required to integrate changed content up to the data store. This is the reason why it is not required that the path to the file object originates at the data store.

3.2.5 Examples

To illustrate the source concept in action, we will introduce three use cases that deal with different forms of data.

**Data on a filesystem** We assume that there is a directory in the filesystem that holds a collection of music files. Most file formats for music files have the ability to store metadata about the track directly within the file, for example ID3 tags in MP3 files. The adapter graph of a source for this scenario could be composed of the following components:

- A `FilesystemDataStore` that bridges to the actual file system location and returns filenames as records,
CHAPTER 3. NODE ARCHITECTURE

Figure 3.3: Adapting data on filesystems

- a *FileOpenAdapter* that adapts the file names to file objects,
- a *FileStatAdapter* that adapts the file names to file stats, like modification and creation time and
- a *ContentMetadataAdapter* that adapts file objects to the metadata contained within the byte stream. In case of music tracks, this could be Artist, Title, Genre, etc.

The adapter can be composed to the graph shown in figure 3.3. The various output pins of the *ContentMetadataAdapter* are mapped to the type record output, the *mtime* pin of the *FileStatAdapter* is mapped to the anchor and its *size* pin is mapped to the content size output. Finally, the *fileobject* pin of the *FileOpenAdapter* is mapped to the file object output. In this configuration an update of the metadata and content is triggered by the modification time-stamp from the filesystem.

**Data in a database** In this scenario, all information is stored in a database. For example, we can think of a document management system that stores documents (the content) and their associated metadata (like name, author, associated project or customer) in a MySQL database. The components of the adapter graph could be:

- A *MySQLDataStore* that bridges to the database and returns table rows as records,
CHAPTER 3. NODE ARCHITECTURE

Figure 3.4: Adapting data in databases

- a **BLOBAdapter** that adapts binary large objects (the document) from the database to file objects and

- a **MD5HashAdapter** that adapts lists of file objects or strings to hash values.

The assembled adapter graph is shown in figure 3.4. The output pin of the **DataStoreAdapter** that identify metadata are directly mapped to the type record output. Changed records are detected by mapping the output pins of the **MD5HashAdapter** to the anchor output. The file object and content size outputs are fed by the data stored in the available at the **size** and **fileobject** pin of the **BLOBAdapter**.

**Data in a database and content on a filesystem** The third example uses both database and filesystem as backing store. This could be the case for an image library that saves its metadata, title, rating, like tag cloud or image description in an SQLite database. The actual content (the images) is stored in the filesystem and the database stores the location of the file in a field of a data row. The following components could build up the adapter graph:

- A **SQLiteDataStore** that bridges to the database and return table rows as records,

- a **FileOpenAdapter** that adapts file names to file objects,
CHAPTER 3. NODE ARCHITECTURE

Figure 3.5: Adapting data in databases and content in filesystems

- a FileStatAdapter,
- a MD5HashAdapter and
- a ContentMetaDataAdapter.

The type record output gets its data from pins of the DataStoreAdapter directly (external metadata) and from output pins of the ContentMetaDataAdapter (internal metadata). Content is made available on the file object output using the FileOpenAdapter and changes are detected again by using the MD5HashAdapter. This time, the hash is not built over the content, but between a combination of image file modification time-stamp and different fields from the database. This is much faster, as it is not required to build hashes over a potentially big files. Figure 3.5 is the resulting adapter graph.

3.3 Communication Infrastructure

We will now cover the XML-RPC / HTTP, Remote Node Server and Remote Node layers in figure 3.1. We already alluded to two distinct communication channels (section 2.4.1). The first one is dedicated to the actual communication protocol including the exchange of metadata. Channel two is responsible for transferring the actual data.
3.3.1 Control and Metadata Channel

Nodes communicate with each other using a stateless synchronous request-response protocol. Every node exposes the same remote procedure call (RPC) interface that serves as communication foundation. We use XML-RPC over HTTP as an underlying protocol, which provides a simple as well as a platform independent solution. The main purpose of this channel is to exchange metadata in the form of *meta records*.

The server part of XML-RPC is implemented by the Remote Node Server, that listens for incoming connections and handles requests by invoking the registered handler for that call. Handlers are always executed in a separate thread and nodes are therefore able to serve requests from different nodes concurrently. That poses no problem since we do not support consistency between distinct records, as we saw in section 2.6. The client interface is available through the Remote Node, that encapsulates a Peer. Peers are maintained by the Peer Manager as we will see in section 3.4.2. Both client and server parts share a serialization mechanism that allow transparent transmission of complex structures, even supporting cycles.

### Table 3.2: Fields in the HTTP header used for content transfers.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>content-type</td>
<td>Content type as defined in the HTTP standard</td>
</tr>
<tr>
<td>content-length</td>
<td>Content length as defined in the HTTP standard</td>
</tr>
<tr>
<td>content-checksum</td>
<td>Checksum of the content being sent in the body</td>
</tr>
<tr>
<td>content-version</td>
<td>Version vector that determines the content version</td>
</tr>
</tbody>
</table>

3.3.2 Content Transfer Channel

The control channel is designed to transfer relatively small amount of data at a high level of abstraction. The content transfer channel, on the other hand, uses the HTTP protocol to efficiently transfer potentially large amounts of content from one node to another. We choose HTTP, because it provides a feature to transmit additional header fields separated from the body and does not require to encode binary data for transmission. Only a few extra information besides the payload is sent as part of the HTTP header, summarized in table 3.2.

3.4 Other Subsystems

We now discuss the functions of the remaining components in figure 3.1.

3.4.1 Source Manager

A *Source Manager* is responsible to manage the set of sources (section 3.2) that belong to the node and to detect changes at each source. For this purpose, it
CHAPTER 3. NODE ARCHITECTURE

runs a process that periodically compares the anchor (section 3.1) of each record against the previous version. If the anchor is different, the record is assumed to be modified. The Source Manager will then process the whole source and update the meta- and content record in the Store Manager. The content record is only updated if the size or the hash of the content changed between the current and the previous version. If they are the same, however, the system assumes that only the metadata changed and will only update the meta record. The updated meta record will then point to the unchanged content record. The obvious advantage is that it not required to transfer the whole content if it has not been changed, which is useful in the case that the content transfer protocol does not implement a differential transfer method.

Secondly, Source Managers provide a high level interface for the rest of the system, that allows to fetch and change content as well as change meta records without knowing anything about the underlying sources. For this reason, they maintain mappings between IDs of records at specific sources and universal unique identifiers (UUID) that have been assigned when records first appeared in the system. Details about ID handling is covered in section 5.2.

The Source Manager also provides a facility to look up other sources based on their names. This is required for records that reference records from another source, which is represented through the source adapter in the adapter graph.

There is currently no support for a notification mechanism at source level. For filesystem data stores, a notification system like inotify, which is available for recent Linux kernels, could be used. In case of database data stores, PostgreSQL, for example, provides a listen/notify SQL extension. Combined with a trigger, that is registered at the database when a source is added to the Source Manager, a notification mechanism could be built for this specific DBMS. When it comes to MySQL, there is currently no clean solution: even though MySQL supports triggers, they can only contain SQL statements. There is a work in progress to support external language stored procedures 5 which would be a possible approach for a custom notification mechanism once it has been completed. SQLite on the other hand, uses a single file as storage for the database, so file notification could again be used to at least reduce the polling frequency. Because of the variety of custom solutions required, we decided not to implement source notification for the prototype. Instead we will discuss how polling performs in section 5.7.

3.4.2 Peer Manager

As soon as a node is started, it periodically broadcasts packets that act as a beacon containing both its ID 6 and the communication port 7 it is listening on. Other nodes will eventually pick up the beacon and try to establish a connection. If the transport layer does not support broadcasting mechanisms or if the nodes are located in different broadcast domains, but within a single network, this cannot work. Instead of using a central dictionary service, the API of the node has to give hints in form of addresses, where tables are reachable at.

5http://forge.mysql.com/wiki/ProjectPage_External_Language_Stored_Procedures
6Section 5.2 covers node identifiers in detail.
7The implementation uses POSIX sockets as transport layer, so this is actually a TCP port. The port may vary, if the specified base port was already in use by another application (or by another node instance).
This information will be stored persistently, so that the hint has to be given only once (as long as the address does not change).

The Peer Manager realizes this discovery mechanism and manages the resulting set of peers. Connection and disconnection events of peers are made available through an event subscription interface to the rest of the system. It can be configured by the following parameters:

**Discovery interval** defines the time between the two discovery beacon broadcasts. Beacons detect yet unknown nodes within the same broadcast domain.

**Heartbeat interval** defines the time between two heartbeats. Heartbeats detect the liveliness of nodes already known in the system. A smaller value will increase the frequency at which nodes are detected, but also increase the network traffic.

**Peer timeout** defines the time after which a node is considered to be disconnected, if no message has been received. This value should be higher than the heartbeat interval.

Peers are represented through peer objects that allow to assign persistent addresses to them. A persistent address is an alternative location where a node can be reached at. This is useful for cases where a node is located in a different broadcast domain but at an address that rarely changes.

### 3.4.3 Subscription Manager

The mapping between nodes and their subscribed filter is under control of the Subscription Manager. It also allows to query which filters match for a specific record or to conveniently publish subscriptions to other nodes.

### 3.4.4 Store Manager

We already talked about meta and content records and it is obvious, that they need to be stored persistently somewhere. This is the job of the Store Manager. A Store Manager provides two distinct stores, one for meta records and one for content records. Every store has the ability to list, create, get, update or delete records, as well as to acquire and release read or write locks for specific records. The locks provide a way to synchronize concurrent accesses from different subsystems.

Another task of the Store Manager is to determine where remote data (see section 2.2.2 and 2.4) is stored to. An API allows to define mappings between filters and storage locations and provides a default mapping for filters without an explicit mapping.

---

8 An alternative to the automatically discovered address.
9 Not to be confused with data stores of sources.
10 Also known as shared and exclusive locks.
11 In the sense of access synchronization, not data synchronization.
3.4.5 Sync Manager

The Sync Manager implements the synchronization protocol we discussed in section 2.5. Connected peers are immediately notified by the Sync Manager about changed records that match at least one partial or full subscription. Additionally, if content has been changed, a content notification is sent to all connected peers that match at least one full subscription for this record. If a record is deleted, it is not possible to apply a filter to it, because the associated type record is not available anymore. A deletion notification is therefore sent to all connected peers, which is acceptable as it happens only once. Upon receipt of a notification, the Sync Manager determines the effect of the updated meta record and updates the meta record if applicable. In case of a conflict, an event will be fired that allows the user to take the appropriate action.

Detecting and updating outdated content also lies in the responsibility of the Sync Manager. We first need to have a definition of outdated content. Firstly, every content record has a version and meta records keep track of the content version they are valid for. Consequently, content is outdated, if the version of the content record does not match the content version stored in the meta record. This makes it possible to distribute changes that apply to the meta record only without retransmitting the whole content. Secondly, each content record is either in the state valid or locked. A record moves from the valid to the locked state if the file that stores the content was changed and the change has not been committed yet. Locked content records are never considered outdated and are therefore neither replaced by remote copies nor sent to other nodes.

The test if a single content record is outdated happens when a content notification is received and after an explicit sync, as well as periodically for all records on the node. After detecting outdated content, the Sync Manager queries other peers for a version of the content that matches the one referenced in the meta record. The strategy how data is transferred is not explicitly defined: in the simplest case, the whole content is fetched from a single node whereas a more sophisticated protocol could fetch multiple chunks of data from different nodes concurrently. If content has been received, that is not yet known to the node, the storage location is determined by querying the Store Manager. The file name, on the other hand, is inferred by the name of the record, the originating node and the MIME type. For example, an image that was received from the node named Cellphone with MIME type 'image/png' and image name 'Company Logo' could get the file name 'Company Logo (from Cellphone).png'.

If modified content is received at the node that created the record, it is not stored in the filesystem, but has to be integrated via the source to its original location. We can not guarantee that other nodes have updated the type record so that it matches the modified content, as the definition of how to extract metadata is local for every source. It is therefore possible that the integration of the content can lead to an updated type record. After integration, the type record is adapted and in case it changed, the meta record is updated accordingly. It will then be distributed to other nodes again, but since it still points to the same content record, the existing content on the other nodes will be recognized as up-to-date and additional data has to be transmitted.

\footnote{Actually, the notification is deferred a short while. The underlying mechanism is discussed in section 5.5.}
3.4.6 Persistent Property Manager

To store data persistently the *Persistent Property Manager* allows other modules to save their configuration persistently. Some examples for persistent configuration are the nodes name (in case of a node), the subscribed filters (in case of the *Subscription Manager*), or the refresh cycle timings (in case of the Sync Manager). Initialization and access to the *Persistent Property Manager* is provided via the *Node*. 
Chapter 4

Interfaces

This chapter explains some important interfaces of the system and how they are used. The first part is about the user API, represented as dotted double-arrow in figure 2.1 whereas the second is about the node to node interface that is used for communication between the nodes.

4.1 User – Node API

The user-node API allows the user of a device to control the synchronization framework.

4.1.1 Node API

The node API is the entry point for the user to control the system. It allows to access a node’s properties and functions as well as the other subsystems API. We will now give an overview of the nodes API, grouped in categories.

System information

To query information about the node, the API provides access to

- the configuration directory location `getConfigDirectory()`,
- the nodes UUID `getUUID()`,
- the port of the socket it is listening on `getPort()`,
- the peers, either connected only or all known `getPeers()`,
- the logical time-stamp `getTime()` and
- the human-readable name of the node `getName()` and `setName()`.

Query data

To get the UUIDs of all records in the system, the `getRecordUUIDs()` method is used:


```python
def getRecordUUIDs(self):
    """Returns all the UUIDs of all records in the system."
    """

    The meta record, that also contains the metadata about the content is fetched using `getMetaRecord()`:

def getMetaRecord(self, uuid):
    """Returns the meta record with a specific UUID.

    - uuid: The UUID of the record to get.
    Returns None if the record is not known.
    """

To explicitly fetch content, either because a record is only subscribed partially, or a fully replicated record is not yet up-to-date, the `fetchContent()` method can be used:

def fetchContent(self, uuid, peer=None):
    """Fetches the content of a record.

    - uuid: The UUID of the record whose content to fetch.
    - peer: Optional peer to fetch the content from. A peer is chosen automatically if omitted.
    """

    The last synchronization result of a record can be queried by `getLastSyncResult()`:

def getLastSyncResult(self, uuid):
    """Returns the last synchronization action of a record.

    - uuid: The UUID of the record to get the last sync result from.
    Returns None if there is no synchronization result available.
    """

This method also provides records with conflicting state. But since querying conflicting records is a frequent operation, there is a dedicated method `getConflicts()`:

def getConflicts(self):
    """Returns a list of all pending conflicts.
    """

Both methods return a `ProcessRecordResult` object that contains both meta records that were involved in the sync and the sync result.

**Update data**

One of the most important operation of the node API is the `commit()` method, which is used to update the system with changed remote data:

def commit(self, uuid, typeRecord=None, file=None):
    """Commits changed content to the system.

    - uuid: The UUID of the record to commit.
    - typeRecord: An optional typeRecord, that may contain changed
    """
```
metadata. If it is omitted, the previous typerecord will be used.
- file:
  An optional file where the content is stored at.
  If omitted, the previous location will be used.

For optimistically replicated records, this will always succeed,
but may produce diverged copies which require a conflict resolution
later.

Returns True if the commit succeeded, False otherwise.

If a synchronization has led to a conflict, it has to be resolved. This can be
done by calling resolveConflict():

def resolveConflict(self, uuid, resolution):
    """Resolves a pending conflicts.
    - uuid:
        The UUID of the record that is conflicting.
    - resolution:
        One of the following values:
        - local
        - remote
        - merge

    Does nothing if the record was not conflicting.
    The result can be verified using the getLastSyncResult() method.
    ""

To delete data completely from the system, the purge() method can be
used. Usually, the subscription for the filter should be canceled first, else the
data would reappear soon again.

def purgeRecords(self, filter):
    """Purges all records (metadata and data) from the node
    that matches filter.
    - filter:
        The filter that is used to select the records to be purged.
    ""

Data Subscriptions

By subscribing a filter, nodes start to receive data:

def subscribe(self, filter):
    """Subscribes for records that match a filter.
    - filter:
        The filter to subscribe.

    A record is considered as subscribed if it is matched by at
    least one filter.
    ""

It is possible to remove a subscription again, of course:

def unsubscribe(self, filter):
    """Unsubscribes a filter.
    - filter:
        The filter to unsubscribe.
    """
CHAPTER 4. INTERFACES

Maintenance

If a node is not reachable within the same sub-net, the auto-discovery mechanism will not work. In this case, the user may provide the node with address hints:

```python
def giveAddressHint(self, address):
    """Gives a hint, that a node can be reached through an address.
    """
    address:
    A (host, port) tuple.
```

Even if storage location for remote data changes, content is not automatically moved. If desired, the user can do that either for a specific record or for all records:

```python
def relocateContent(self, uuid):
    """Relocates existing content of a record to the location that is currently mapped by the store.
    """
    uuid:
    The UUID of the record to relocate.
```

```python
def relocateAllContent(self):
    """Relocates all existing content to the locations that are mapped by the store.
    """
```

The user may invalidate content, so that it is downloaded again:

```python
def restoreContent(self, uuid):
    """Restores the content by fetching it again from another peer.
    """
    uuid:
    The UUID of the record whose content is to be restored.
```

Subsystems

It is also possible to get direct access to the APIs of the following subsystems: Persistent Property Manager, Store Manager, Source Manager, Peer Manager, Subscription Manager and Sync Manager. This way, the user has full control over the system, even though it is usually sufficient to use one of the convenience-functions we just presented.

4.1.2 Source API

The source API allows to assemble adapters to sources as well as to adapt and integrate data using the resulting graph. For the user only the first is relevant, as the second is commonly done by the Source Manager and Sync Manager automatically.

Building a source always involves six steps:

1. Create a source by supplying a name and a type.
2. Create a data store and configure its options.
3. Assign the data store to the source. The outputs of the data store are then available from the source as if it was an ordinary adapter.

4. Create adapters by supplying a name and optionally configure their options.

5. Add every adapter to the source, by specifying all its input connections.
   An input pin is connected to an output pin by defining an (adapter name, output pin name) tuple.

6. Map adapter output pins to the source outputs.

Listing 4.1 are the calls required to set up the adapter graph shown in figure 3.4.
Note that the components were not named in the figure and that the select query in the listing has been slightly extended to include the content type.

Steps 1 to 4 are self-explanatory, but we will shortly discuss a few aspects relevant for steps 5 and 6. Besides the adapter itself, the addAdapter() method accepts a variable-amount of arguments that define the inputs of every pin of the adapter. In contrast to adapter outputs, where every pin is identified by a name, the input pins are identified by the order they are specified in the API.

A single input is defined as a (adapterName, outputPinName) tuple and the method therefore has the following signature:

```python
def addAdapter(self, adapter, *inputs):
    """Adds an adapter object to the source.
    - *inputs:
      A variable amount of (adapterName, outputPinName)
      tuples, that are fed as inputs to the new adapter.
      The inputs are supplied in the order they are specified by the adapter.
    """
```

The inputs of the adapters are specified when they are added to a source, but this is not possible for source outputs, since they are already predefined within the source. The source API therefore provides explicit methods for this purpose. Aside from the type record, all source outputs have only a single input pin, and the outputs are defined by set methods in the form setOutputNameMapping(adapterName, outputPinName). A type record output has multiple input pins, that represent the actual fields in the record. The mapping for the fields are added using the addTypeRecordMapping() method:

```python
def addTypeRecordMapping(self, fieldName, adapterName, outputPinName):
    """Maps an adapter output pin to a field of the type record.
    - fieldName:
      The name of the field in the type record to be mapped.
    - adapterName:
      The name of the adapter used as input.
    - outputPinName:
      Is the name of the pin to be used as input.
    """
```

\footnote{The initial reason for this decision was, that most adapters only have a single input anyway and there are no optional inputs. Defining inputs ordered reflects this property more naturally than using input names, but to be more consistent to the remaining source API, a change might be appropriate.}
# Step 1: Create a new source with a name and a type
s = Source('documentDS', Store.DocumentType)

# Step 2: Create a data store, configure its options
ds = DataStore.MySQLDataStore()

ds.setOption('dbName', 'CMS')
ds.setOption('host', '127.0.0.1')
ds.setOption('dColumnName', 'id')
ds.setOption('user', 'dbuser')
ds.setOption('password', 'foobar')
ds.setOption('selectQuery', 'SELECT id, name, author, mimetype, data FROM docs')
ds.setOption('updateQuery', 'UPDATE docs')
ds.setOption('deleteQuery', 'DELETE FROM docs')

# Step 3: Assign the data store to the source
s.setDataStore(ds)

# Steps 4 and 5: Create adapters and add them to the source
a = Adapters.BLOBAdapter('docData')
s.addAdapter(a, ('documentDS', 'data'))
a = Adapters.MD5HashAdapter('hash')
s.addAdapter(a, ('docData', 'fileobject'))

# Step 6: Map adapter outputs to source outputs
s.addTypeRecordMapping('title', 'documentDS', 'name')
s.addTypeRecordMapping('author', 'documentDS', 'author')
s.addTypeRecordMapping('contentType', 'documentDS', 'mimetype')
s.setFileObjectMapping('docData', 'fileobject')
s.setContentSizeMapping('docData', 'size')
s.setAnchorMapping('hash', 'hash')

Listing 4.1: Creating a source

After the source has been completely built it can be added to the Source Manager, compiles it (section 5.4.1) and makes its content available to the system:

de.getSceneManager().addSource(s)

### 4.1.3 Filter and Subscription API

Filters are implemented as RecordFilter objects, consisting of FilterElements or other RecordFilters. A filter element is a comparison of a single element of a type record against a designated value. The comparison may be an arbitrary function that takes the content of the field of the type record and the expected value and returns a Boolean value. The function has to passed as a string containing Python code:

A filter element that matches the field named “artist” to be exactly “RHCP” is created as follows:

```python
fe1 = FilterElement('lambda fieldContent, value: fieldContent == value', 'artist', 'RHCP')
```

Of course the comparison function can be more complex. For example, it would be possible to compare the color tone of image thumbnails against a threshold or, to stick with the previous example, to do set comparisons:

```python
fe2 = FilterElement('lambda fieldContent, value: fieldContent in value', 'artist', set(['RHCP', 'Radiohead', 'Helden']))
```

See section 6.1.8 for a discussion of the resulting security implications.
The individual filter elements are then assembled to a filter record using the Boolean operators \textit{AND} and \textit{OR}. Because a filter operates on metadata, it has an assigned type and it is only evaluated for type records that match this type. Additionally, a filter can be set as \textit{partial}, which means that the actual data is not automatically fetched. A (non-partial) filter that matches all music tracks released 2004 or after \textit{AND} the filter element \textit{fe2} is created like this:

\[
\text{fe3} = \text{FilterElement( 'lambda fieldContent, value: fieldContent \geq value', 'year', '2004')}\]

\[
fr = \text{FilterRecord(Store.Music, False, fe2)}
\]

\[
fr.appendAnd(fe3)
\]

For flat composition of filter elements the common binding rules of Boolean algebras apply during evaluation of a filter. To group evaluation, record filters can also be nested. If we denote \textit{fe4} as another filter element, a nested record filter that is evaluated as \((\text{fe2} \lor \text{fe3}) \land \text{fe4}\) is built as follows:

\[
\text{frSub} = \text{FilterRecord(Store.Music, False, fe2)}
\]

\[
\text{frSub.appendOr(fe3)}
\]

\[
fr = \text{FilterRecord(Store.Music, False, frSub)}
\]

\[
fr.appendAnd(fe4)
\]

The last step is to subscribe the final filter using the node API:

\[
\text{node.subscribe(fr)}
\]

\section*{4.2 Node – Node Interface}

\subsection*{4.2.1 Remote Node Interface}

Nodes communicate with other nodes by accessing their remote interface through \textit{RemoteNode} objects. All operations are synchronous and the caller is therefore blocked until a response is received. Because the protocol is stateless, most operations require the node to send its UUID, which is not mentioned in the methods description. This applies to the \textit{map} parameter that maps temporary short IDs to UUIDs as well. If the map is not specified, the methods expect uncompressed UUIDs.

Nodes use the \textit{subscribe()} method to subscribe their filters to other nodes:

\[
\text{def subscribe(self, nodeUUID, filters):}
\]

\[
"""Subscribe this node to be notified for changed records that match at least one filter."
\]

\[
- filters:
 List of filters.

Previous subscriptions are deleted.
"""

If required the filters can also be polled:

\[
\text{def getSubscriptions(self, nodeUUID):}
\]

\[
"""Returns this nodes subscription for a node with UUID nodeUUID."
\]

To notify other nodes about changed records, \textit{notify()} and \textit{notifyContent()} are used:
def notify(self, nodeUUID, metaRecords, map = None):
    """Notifies this node about changed meta records.
    - metaRecords:
      A list of changed meta records.
    """

def notifyContent(self, nodeUUID, recordUUIDs, map = None):
    """Notifies this node about changed content.
    - recordUUIDs:
      A list of UUIDs of records that have their content changed.
    """

Nodes announce themselves using heartbeats:
def heartbeat(self, nodeUUID, serverPort, name, type):
    """Announces a node and updates its address if necessary.
    - serverPort:
      The port the node listens for RPC requests.
    - name:
      The (human-readable) name of the node.
    - type:
      The address type, either 'persistent' or 'auto'.
    """

A one-way synchronization is executed by calling sync():
def sync(self, nodeUUID, syncVVs, map = None):
    """Returns the changed metarecords based on the received synchronization vectors.
    - syncVVs:
      The synchronization vectors that are used to determine changed meta records.
    """

To explicitly commit a meta record, commit() can be used:
def write(self, nodeUUID, metaRecord, map = None):
    """Commits a changed meta record.
    - metaRecord:
      The meta record to commit.
    """.
    Returns True on success, else False.

Meta records can also be retrieved explicitly:
def fetchMetaRecord(self, recordUUID):
    """Fetches a meta record.
    - recordUUID:
      The UUID of the record to fetch.
    """

To examine the status of content on a node, fetchContentResponse() is used:
def fetchContentResponse(self, recordUUID):
    """Fetches the content response of a record.
    - recordUUID:
      The UUID of the record whose status is to fetch.
    """
CHAPTER 4. INTERFACES

<table>
<thead>
<tr>
<th>Code</th>
<th>Reason</th>
<th>Long description</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>OK</td>
<td>Content is valid and can be fetched.</td>
</tr>
<tr>
<td>204</td>
<td>Content unavailable</td>
<td>The requested record is known, but the data is currently not available.</td>
</tr>
<tr>
<td>404</td>
<td>Content not found</td>
<td>No record with the requested UUID exists.</td>
</tr>
</tbody>
</table>

Table 4.1: HTTP content request response codes

Content is fetched using `fetchContent()`. This returned response object is the same as in the previous method, but the actual content is transmitted as well.

```python
def fetchContent(self, recordUUID):
    """Fetches the content of a record."
    recordUUID:
        The UUID of the record whose content is to fetch.
```

4.2.2 Fetch Content Interface

Even though the remote node interface also allows to access content the same way as other operations are executed, content can also be requested through a simple HTTP request, whereas the record UUID is used as path in the URI. The HTTP server listens on the adjacent port of the nodes RPC server and supports HTTP `GET` and `HEAD` requests that it responds with the codes shown in table 4.1. The response to a `HEAD` request are the fields listed in table 3.2, only, without any payload, which allows a node to decide if the remote content matches the version it expects. Only in a second step, the actual data is fetched by a `GET` request.
Chapter 5

Prototype Implementation and Evaluation

In this chapter we will give insights on the implementation of the system prototype. We start with a short overview followed by the discussion of the version vector compression. The implementation of the adapter graph processing, the delayed event dispatch and the concurrency handling are followed by a performance evaluation that closes the chapter.

5.1 Overview

The prototype is implemented in Python. The decision for Python was because it provides a rich set of libraries with bindings to a lot of other languages and applications. Since the Python interpreter is available on almost all platforms, it makes it very easy to run the system on different platforms. Of course, the architecture is modular and the components can be replaced with another implementation as long as the modules API is maintained. For example, nodes provide a set of operations through a remote node API with a synchronous semantic. The serialization and RPC mechanisms are completely hidden and can be exchanged as long as this semantics is preserved. Another example is the store for content and meta records. The current implementation uses Python shelves. Shelves are persistent, associative arrays, that can store arbitrary Python objects. Because shelves are not required to be adjusted when the data structure to be stored changes its model, they make rapid prototyping possible. In a real implementation, shelves would most probably be replaced by either an well-known database system or a custom persistency mechanism. Talking about shelves and persistency: the Persistent Property Manager is also backed by a shelf.

The node component manages all modules and every module has a reference to its managing node. This way the modules can access services provided by other modules if required. But most interaction between the modules is implemented using events: modules allow to register event handlers (or listeners) which reduces the direct coupling between them - it is rather delayed to the initialization routine of a node that registers the various handlers. Another advantage is that the user of the framework can have influence on the functionality
of the framework by intercepting events to their own handlers.

Some modules come in the form of managers as discussed in section \ref{sec:management}.

Some managers provide \texttt{start} and \texttt{stop} operations that launch and terminate control loops running in a separate thread. The \texttt{Peer Manager}, for example, runs four such loops that send and handle discovery beacons, heartbeats and detect non-peer timeouts, whereas the single loop in the \texttt{Source Manager} periodically scans all sources for records that have been changed.

\section*{5.2 Identifiers}

To uniquely identify a node within the system it requires an ID that only exists once. When a node is started for the first time, it generates a Version 4 UUID and stores it persistently. A UUID is a 16-byte number and Version 4 are UUIDs that are generated randomly. The bits 76-79 contain the fixed value 0100 (4) that specifies the version number. Additionally, the bits 66-67 are reserved and contain the value 01. This means from the whole 128 bits, 122 can be freely allocated, which results in $2^{122}$ different possible UUIDs. If we denote $n$ as the number of nodes in the system, the probability that a collision \footnote{A collision occurs if at least two nodes in the system generated the same UUID.} occurs is

$$p(n) = 1 - \prod_{k=1}^{n-1} \left(1 - \frac{k}{2^{122}}\right)$$

and the expected number of nodes in the system before two generate the same UUID is $2^{122}/2$. For every feasible number of $n$, the probability of a collision can therefore be considered as near 0.

Records are identified by their create version vector, which consists of the UUID of the node and the time counter when the record was created. The record ID is then generated by concatenating those two values. The definition of record IDs by version vectors make them applicable for the same compression method as we will use for version vectors.

\section*{5.3 Compressing Version Vectors}

The largest component in every version vector is the UUID of the node. Every meta record contains three version vectors, whereas the sync version vector may contain UUIDs for every node in the system in the worst case. If UUIDs are stored as integers, at least 48 bytes are required per record. To reduce this data, a node maps encountered UUIDs to consecutive numbers, starting by 0. It then replaces every UUID in the meta records with the mapped value and saves the mapping persistently. It is also possible to apply the same idea to reduce the amount of data during data transmission: the sender just has to send the mapping first. If a single byte is used as a replacement, the amount of data can be reduced up to a factor of 16 for a maximum of 256 nodes.

Figure \ref{fig:compression} visualizes the compression ratio for four different numbers of nodes in the system depending on the number of records on a node for the case that every meta record knows all nodes in the system.
5.4 Adapter Graph Processing

5.4.1 Source

In section 3.2.2 we explained that there are two type of adapters. Read-only adapters are instances of AbstractAdapter, that define among other things the method `adapt`, that has the following signature:

```python
def adapt(self, input):
    """Adapts inputs to outputs.

    input is an associative array (pinName->input value)

    Returns an associative array (pinName->output value)"
```

`adapt` therefore implements the actual functionality of the adapter. As we see, data is passed around through associative arrays, that contain mappings from the names of the pins to the values.

Read-write adapters are subclasses of AbstractAdapter and therefore instances of ReadWriteAdapter. They define the two additional methods `integrate` and `prepareIntegrate`:

```python
def integrate(self, changedOutput, input = None):
    """Integrate changed output to input.

    changedOutput is an associative array (pinName->output value)
    input is an optional associative array (pinName->input value)

    Returns an associative array (pinName->input value)"

def prepareIntegrate(self, input):
    """Prepares input value for integration.

    input is an associative array (pinName->input value)

    Returns an associative array (pinName->input value)"
```

The `integrate` method is just the opposite operation to `adapt`. Using `prepareIntegrate`, adapters can optionally calculate additional inputs that are re-
CHAPTER 5. PROTOTYPE IMPLEMENTATION AND EVALUATION

```python
def addAdapter(self, adapter, *inputs):
    """Adds an adapter object to the source.
    Inputs is a variable amount of (adapterName, outputPinName) tuples, that are fed as inputs to the new adapter.
    The inputs are supplied in the order they are specified by the adapter.
    ""
    self.adapters[adapter.getName()] = adapter
    dependencies = {
        i = 0  # Output number (inputs are ordered!)
        for t in inputs:
            # Store output dependencies
            if self.adapterOutputs.has_key(t[0]):
                self.adapterOutputs[t[0]][t[1]] = (adapter.getName(), i)
            else:
                self.adapterOutputs[t[0]] = {
                    if self.adapterOutputs[t[0]][t[1]] == (adapter.getName(), i):
                        self.adapterOutputs[t[0]][t[1]] = (adapter.getName(), i)
            dependencies[i] = t
            i = i + 1
    # Store the input dependencies
    self.adapterInputs[adapter.getName()] = dependencies
    return
```

Listing 5.1: Adding an adapter to a source

required for integration. We will see why this is necessary for some kind of adapters after we discussed how adapters are composed and evaluated.

Adapters are added to a source by specifying their inputs in terms of other adapter and output pin names. This relation can be understood as a set of dependencies between nodes that describe a directed acyclic graph (DAG). Because we do not only want to process information from the root to the leaf but also backwards, the DAG has to be built for both direction. We define two mappings `adapterOutputs` and `adapterInputs` that map each pin of an adapter to the adapter and pin name the data comes from. The `adapterOutputs` structure stores the dependencies to adapt data (forward processing) and the `adapterInputs` structure the dependencies for integration (backward processing), respectively. These data structures are therefore associative arrays whose keys are adapter names and whose values are again associate arrays from pin names to tuples consisting of adapter name and pin number: `adapterName → (pinName → (adapterName, pinNumber))`. The current API does not specify names for the input pins of each adapter. Instead, pins are ordered and the API requires to specify input dependencies in this order. The two maps are set up using listing 5.1. Besides these dependencies, an additional associative array `adapters` maps adapter names to adapter objects is built. This mapping is later required for further processing.

The input argument in line 1 accepts a variable-length amount of (adapterName, pinName) tuples that define the inputs of the specific pin at that position, as we have seen in section 4.1.2.

After all adapters have been added to the source, the source has to be compiled. In this step a DAG is built first for both direction of processing. The DAG defines a partial order between the adapters, which we can use as an order in

---

2 It is always possible to define a single implicit root node that all other root nodes depend on.

3 Error handling is omitted in all listings.

---
def _compile(self, dependencyMap):
    """Returns a DAG and its serialized form, specified through a set of dependencies."
    dag = {}
    for k, v in self.adapters.items():
        dag[k] = DAGNode(v)
    root = DAGNode(None)
    for depName, dependent in dag.items():
        dependency = None
        if dependencyMap.has_key(depName):
            for dependentPin, DependencyTuple in dependencyMap[depName].items():
                dependencyName = DependencyTuple[0]
                dependency = dag[dependencyName]
                dependency.addChild(dependent)
    if dependency is None:
        root.addChild(dependent)
    # Walk through DAG and calculate weight of nodes
    self._calcWeight(root, 0)
    # Create a total order by sorting by weight
    result = [node.getObject() for node in sorted(dag.values(),
        lambda x, y: x.weight - y.weight) if node.weight >= 0]
    return dag, result

Listing 5.2: Calculation of the DAG and its serialized version

which we evaluate every adapter. The _compile function calculates a DAG and a total order out of the dependency maps from the previous step and is shown in listing 5.2. In lines 6 and 7 a DAGNode object is created for every adapter. DAGNodes store references to their children and parents, which are populated from the dependencies in lines 10 to 20. A distinguished root node is created in line 10 and every adapter that does not have any input dependencies is linked to this root in line 20. A depth-first search (listing 5.3) in line 23 calculates the weight of every node in the DAG which is used as sorting criterion to build a total order (of adapters, not node objects) in line 26. The weight is defined as the longest path from the root to the node. The depth-first search algorithm uses a very simple criterion to detect cycles in the graph: if the depth of a node is greater than the total amount of nodes, there must be a cycle. This test is acceptable in our case, because we only have a small number of nodes. As a result we get two DAGs, adapterNodesForward and adapterNodesBackward, and two sequences of adapters, forwardOrderedAdapters and backwardOrderedAdapters. Except for the adapterNodesBackward, they are used in the subsequent steps. If we evaluate every adapter of the source in the order they occur in forwardOrderedAdapters (for adaptation) or backwardOrderedAdapters (for integration) we know that all inputs that are required for a certain adapter are already available, because its dependencies have a weight that is strictly less and are therefore already processed.

The next step is to calculate subsets of all adapters that are required to process the graph. Table 5.1 lists all sets that are required. In case of adaptation all nodes that are on a path from the root to a specific output have to be processed and the rules for integration were covered in section 3.2.4.

Listing 5.4 is the algorithm that determines the set of nodes that are required
for processing by talking three arguments: a mapping that defines the leaf adapters, a pathNodeFilter that is used to describe constraints that filters on a path are required to fulfill and a targetNodeFilter that specifies the target node of processing. Both filters are optional, in which case there is no restriction on the nodes on the path or the target is an arbitrary leaf node, respectively.

The algorithm maintains two sets: a set of marked adapters that identify the adapters that apply to the requirements, and a set of visited DAG nodes, that avoids visiting a single node more than once. The sets are populated in the recursive function markDAGNodes that walks backwards and depth-first through the DAG. This function is called once for every node in the DAG (lines 40-43). If the node was already visited (through another path in the DAG) it immediately returns (10-11). Otherwise, the node is added to the set of visited nodes and further examined (14).

If a pathNodeFilter was specified and the adapter in inspection does not match that filter (17), the path has to end here. This means, the current node has to be either a valid target node, in which case we add the node to the set of marked nodes (19 or 22), or we did not reach a target and the whole path is abandoned. In both bases, we have finished inspecting the path and return (23).

If the current node is a valid node in the path and also a target node (25-26), we mark it and return. Only if none of those cases apply, we start inspecting the parent nodes (30). The current node is part of the path, if and only if the parent node is marked after the recursion (35-36), in which case it is marked again.

After all nodes have been visited, all marked adapters are returned (45). They now define a subset of all nodes that are required to be processed. The processing rules can now be encoded into inputs to this function as shown in table 5.2.

The differentiation between the different subsets is important because as we saw in section 3.2.4 in case of integration not all adapters must be processed. It is also important to maintain a good performance: some adapters require lengthy calculation, for example to extract metadata from files or to create thumbnails of images. If such calculations had to be done for every refresh cycle, it would usually be too expensive, at least for most portable devices. That problem can be avoided by making sure that only inexpensive adapters
def _getDependentNodes(self, mapping, pathNodeFilter = None, targetNodeFilter = None):
    """Returns the nodes in this source DAG that are required to process data from or to outputs defined by the parameter mapping.
    """
    marked = set()  # Contains adapters
    visited = set()  # Contains DAGNodes
    def markDAGNodes(dagNode):
        if dagNode in visited:
            return
        dagObject = dagNode.getObject()
        visited.add(dagNode)
        if not pathNodeFilter is None:
            if not pathNodeFilter(dagObject):
                if targetNodeFilter is None:
                    marked.add(dagObject)
                else:
                    if targetNodeFilter(dagObject):
                        marked.add(dagObject)
            return
        if not targetNodeFilter is None:
            if targetNodeFilter(dagObject):
                marked.add(dagObject)
            return
        for parent in dagNode.parents:
            # Skip the (pseudo) root adapter
            if parent.getObject() is None:
                marked.add(dagObject)
            else:
                markDAGNodes(parent)
                if parent.getObject() in marked:
                    marked.add(dagObject)
        return
    for m in mapping:
        adapterName = m[0]
        adapterNode = self._adapterNodesForward[adapterName]
        markDAGNodes(adapterNode)
    return marked

Listing 5.4: Determining dependent nodes

<table>
<thead>
<tr>
<th>Set name</th>
<th>Operation</th>
<th>Evaluation order</th>
</tr>
</thead>
<tbody>
<tr>
<td>typeRecordAdaptation</td>
<td>Extract the type record from source record</td>
<td>forward</td>
</tr>
<tr>
<td>typeRecordIntegration</td>
<td>Integrate a modified type record back into a record</td>
<td>forward / backward</td>
</tr>
<tr>
<td>anchorAdaptation</td>
<td>Extract the anchor value</td>
<td>forward</td>
</tr>
<tr>
<td>sizeAdaptation</td>
<td>Extract the size value</td>
<td>forward</td>
</tr>
<tr>
<td>contentAdaptation</td>
<td>Extract the size and file object values</td>
<td>forward</td>
</tr>
<tr>
<td>contentIntegration</td>
<td>Integrate a modified file object back</td>
<td>forward / backward</td>
</tr>
</tbody>
</table>

Table 5.1: DAG processing subsets
are used to build the anchor value.

After the compilation is finished, which is of course only required once for every source, the actual evaluation of the adapter graph for adaptation operations is straightforward: every adapter in `forwardOrderedAdapters` that is contained in the set of the specific operation is executed in order and the result is stored in a value store. The inputs of the following adapter can then be fed from this store. On the other hand, integration operations require additional care. It is possible that some adapters require data from their inputs to successfully integrate data. For example, the `FileOpenAdapter` requires the file name to integrate changed content back to that file. This is were the `prepareIntegrate` comes into action. It has the same signature as the common `adapt` method, but instead of adapting the inputs, it just returns (or calculates) all inputs that are required for an integration process. The source is then responsible to feed this data to the adapter via the optional input argument of the `adapt` method. An integration process consists therefore of the following steps:

1. An adaptation run, by iterating through the `forwardOrderedAdapters`, with a call of `prepareIntegrate` in addition of the normal `adapt` call.

2. An integration run, by iterating through `backwardOrderedAdapters`, that supplies the additional input to the `integrate` method, if available.

Of course, both iterations only process nodes that are in the set of adapters that represent the operation (see again tables 5.1 and 5.2).

### 5.4.2 Source Cache

From the perspective of a single source, every record is identified by a local ID that is defined by the data store itself. Once such a record is added to the system, it gets a UUID assigned as we saw in section 5.2. To maintain the mapping between the local ID and the UUID, every `Source Manager` has a source cache that maps UUIDs to source cache records. Besides the local ID, they contain the last anchor value, that is used to detect changes to records, and the source name that the record originates from. Because every source cache is responsible for a single node, the node UUID part of the record UUID is the same for all records. Consequently, we do not need to store the whole UUID but instead only the local time counter, which saves at least 16 bytes of storage per record. The amount of storage for source names can also be reduced using
CHAPTER 5. PROTOTYPE IMPLEMENTATION AND EVALUATION

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delaySeconds</td>
<td>The time the event dispatcher waits between an event and its dispatch. If an other event is fired within that time, the scheduled dispatch is deferred for another delaySeconds.</td>
</tr>
<tr>
<td>maxQueuedEvents</td>
<td>The maximum number of event that are waiting to be dispatched. If the queue length at the time an event is fired exceeds this value, the next scheduled dispatch is not delayed any longer.</td>
</tr>
<tr>
<td>maxMerge</td>
<td>The maximum number of events that are merged into a single dispatch.</td>
</tr>
</tbody>
</table>

Table 5.3: Parameters of the delayed event dispatcher

the same method as explained in section 5.3, as there are usually only a few different sources per node, but a lot of records per source. To allow fast look-ups of existing UUIDs, the backwards mapping from source name and local record ID to UUID is also stored in the source cache.

5.5 Delayed Event Dispatch

In addition to a traditional event registration and dispatch mechanism, we implemented a delayed dispatcher that collects events and dispatches them in one batch. The dispatcher can be configured by the parameters listed in table 5.3. By tuning the parameters, the trade-off between efficiency and latency can be controlled. An upper bound for the latency $L_{\text{max}}$ can be estimated as follows, where $l$ denotes the current length of the queue and $\text{dispatchTime}$ the time to dispatch a (merged) event to all listeners.

$$L_{\text{max}} = \begin{cases} 
\text{delaySeconds} \cdot \text{maxQueueEvents} & \text{maxQueueEvents} > l \\
\text{delaySeconds} + \frac{l}{\text{maxMerge}} \cdot \text{dispatchTime} & l \leq \text{maxQueueEvents}
\end{cases}$$

The second case may occur when events are processed slower than they are fired over an extended period of time. The mechanism does not reject new events or block the caller even if the queue will exceed $\text{maxQueueEvents}$. Instead, the size of the queue is increased temporarily and events are dispatched without any delay to reduce the size of the queue as fast as possible.

Delayed event dispatch is especially useful for notification events, because the cost of establishing a connection to a remote host can be distributed among multiple events. For example, if a node connects to another one, each starts a one-way sync which may lead to various updated records. Instead of sending a notification for every change to all interested nodes, they are merged by the delayed event dispatcher sent in a single remote procedure call that contains all changes.
5.6 Concurrency Handling

Concurrent access to data requires a mechanism to either synchronize those access or, if this is not possible, to detect inconsistencies and take appropriate measures.

We will now look at two examples, where concurrency handling is important. The first is about internal data structure, where a node is in full control of. It is therefore possible to serialize access to such structures.

The more interesting case is the example about remote data. Since a node has no exclusive control over this type of data, we show how detection of inconsistencies is implemented.

5.6.1 Internal Data

Two structures where a lot of concurrent access happens are the meta records and content records that hold metadata for every record. To control access to those data structures, a `RecordLock` class provides operations to acquire and release read or write locks on a per record basis. Write locks, also called exclusive locks, can only be acquired once and the owner of the lock has the permission to modify the locked record. Read locks, or shared locks, can be acquired multiple times, but only if the record is not write locked.

The implementation of the `RecordLock` class is straightforward and is built using three components:

- A dictionary that maps UUIDs to a number of active read locks,
- A set that holds all UUIDs of records that are write locked and
- A condition variable that manages modifications to those sets.

The implementation is attached in Appendix C.

5.6.2 Remote Data

In section 2.2.2, we saw that remote data is stored in the filesystem and the user of the system has direct control over that data. On Linux platforms, the ext filesystem does not support mandatory locking, at least not with special mount options. Mandatory locking is a mechanism that allows the same access control as we used for internal data, that is shared, and exclusive locks and no other application can bypass the locking since it is provided directly by the operating system. Linux (and other UNIX flavors) support only advisory locking by default. An application may request locks for file access, but it is not forced to do so. As a consequence, it is therefore possible that the user modifies the files that store remote data while the node reads from it. Since this would lead to inconsistent data being sent to other nodes, we have to detect that situation and handle accordingly.

The problem is handled using a combination of content checksums and filesystem time-stamps: If new content or changes to content is committed by the user, the following steps are executed:

1. Read the modification time-stamp from the filesystem
2. Calculate an MD5 hash value from the whole content
3. Re-read the modification time-stamp from the filesystem

If the timestamps 1 and 3 are identical we assume that the file did not change during the hash calculation. The hash can then be considered consistent and it is stored together with the modification time-stamp as part of the content record. This way, we will detect normal user interaction, but it will not protect from modification time spoofing. On the other hand, spoofing should be considered as an attack, which is out of scope.

Before nodes send content to other nodes, they compare the time-stamp in the content record against the value read from the filesystem. If they match, the data is sent together with the previously calculated hash to the requesting node. After the transmission finished, the modification time-stamp is again read from the filesystem and, in case it changed, the content record is flagged as locked. Marking the record as locked is just an optimization, because from now on, it can be ignored until the modification is either committed or reverted.

On the receiving side, the data is first stored in a temporary file and can be validated by comparing the received hash value with the one calculated from the temporary file. If they do not match, the file will just be discarded. Otherwise, the content record is updated to reflect the new data and the file is moved to its final place.

5.7 Performance

To test how the system performs on mobile devices, we have deployed it on Nokia N810 Internet Tablet devices. They run Maemo 4.1 as operating system, which is a Linux derive for handheld devices. It is equipped with a custom 400 MHz ARM CPU from Texas Instruments with integrated DSP and GPU, has 128 MB of RAM and supports up to 32 GB of storage by means of miniSD memory cards. The operating system already has a Python interpreter, so it has been straightforward to get the framework running. The devices used their built-in wireless LAN capability to communicate with each other.

5.7.1 Source Adaptation Speed

We first measured the compilation time of a source's description to an adapter graph, but it turned out that it was insignificant compared to the actual interpreter start-up time on the tablet. Furthermore, the compilation has to be executed only once for every source, so we focus on tasks that are run often.

A first operation we consider is listing all records available at a source. It only involves listing the IDs of every record, without the actual data. We measured the time required to list a source dependent on the number of records as a mean on multiple runs over the same set of records. This way, the caching mechanisms of the underlying storage technology are in effect and we measure predominantly the processing overhead of the system. Figure 5.2 is a plot of the results for the FilesystemDataStore. As we could expect, the time to list a source's records is related linearly to the number of records. More importantly, even for a few hundred records, the listing is completed in less than a second.

Listing the records on a source is only the first step. Afterwards, the anchor value for every record has to be evaluated to determine if a record was changed. The performance of this process is important, because it has to be done for every
source refresh cycle, even if no data was changed. In section 5.4.1 we saw that the adaptation on a source processes every adapter in the order defined by their dependencies. Consequently, the total time required to adapt a source is the sum of the time to process every single adapter. Since the anchor evaluation time is dependent on the adapter combination we choose, we decided to measure the anchor evaluation time for the two most often used criteria: the modification time-stamp in the filesystem and the hash value of the content, each related to the size of content. We choose content sizes of 1 KiB, 10 KiB, 100 KiB, 1000 KiB and 10000 KiB. The results are plotted in figure 5.3. Note that the x-axis is scaled logarithmically to match the spectrum of inputs. Whereas the modification time-stamp extraction is independent of the file size, we see that using content hashes as anchor can become a bottleneck if large content is processed.

The second very time-consuming adapter is the ContentMetaDataAdapter, that is often used as input for the source’s type record output. We therefore measure the time the ContentMetaDataAdapter requires to extract metadata depending on the type of content for two typical instances of that type. As

Figure 5.2: Source listing time using a FilesystemDataStore

Figure 5.3: Anchor evaluation time
we see in the results shown in figure 5.4 content extraction on the N810 does not perform well. It is interesting though, that the time does not correlate with the size of the content. In fact, it depends heavily on the type of data. This becomes evident when we consider the format how metadata is integrated into actual data. In case of the binary documents (DOC and XLS) the whole structure had to be parsed (and held in memory) to extract the metadata.

5.7.2 Storage Overhead

Besides the actual metadata of the content, the following data is stored for every meta record in the system:

- 3 version vectors
- the size of the content
- the content version
- a flag (that identifies a record as optimistically replicated)

In compressed form (section 5.3), the version vectors occupy roughly 4 bytes each. This also applies for the size (integer value), the flags (integer field) and the content version (which is itself a version vector). In total, we can therefore estimate an overhead of about 16 bytes for every meta record. If we relate this value to the actual payload (the metadata of the content that is), we have an estimated overhead that is in a single-digit percentage range.

For every content on a node, there exists a content record, which stores the following data:

- the size
- the content version
- the checksum
- some flags (local, invalid, locked)
• the location (for remote records)
• the modification time-stamp (for remote records)

As checksum, a 128 bit MD5 hash is used and the location can be an arbitrary length string, which is usually below 200 bytes though. For a content record that represents local data, we therefore have content records with a size of about 28 bytes, and a content record for remote data is rarely bigger than 256 bytes. Compared to the actual data, we have an overhead of much less than 1% for both types of content records.

5.7.3 Communication Overhead

Since we transfer meta records directly in the same serialized form as we use for storage, the same arguments as in section 5.7.2 also apply for communication. In case of content transmission, only the headers listed in table 3.2 are sent in addition to the actual data.

If a node reconnects to another node, it sends its sync version vectors, so that the remote node can decide which records changed. This means data proportional to the total number of records on a node has to be exchanged after a reconnect (or a manual sync), even if only a few records have changed during the disconnected phase. As an optimization, it would be possible to remember the time a node disconnected and only send the synchronization version vectors that were changed since the last disconnect. The remote node would then do the same and only send the meta records that have been modified since the last disconnect (and that are 'newer' than the synchronization version vector the other node sent, if available).

5.7.4 Scalability

The system has been designed to be used on personal devices. There are two main problems that arise in big setups: One is that meta records grow with the number of nodes in the system and never shrink again (see section 6.1.9) and the other is the threaded model of the architecture. Even though we did not run the system in a large scale environment, we assume that the resulting amount of threads, opened sockets and locks would not perform well.
Chapter 6

Conclusion

In this chapter, we will sketch some of the open problems and limitations of the prototype implementation concluded by a summary of the work.

6.1 Problems and Limitations

Unfortunately, the system has some significant drawbacks that disqualify it from a real-world usage whereof some are presented in this section.

6.1.1 Updating Database Data Sources

The current implementation of the SQLDatabaseStore and its subclasses use a simple query rewriting algorithm that generates an update query out of a base query supplied by the user. This works well for queries on single tables, but fails if the query is more complex, like a JOIN query. To support updates to a database that have been adapted with such queries, we could either extend the data store options for such specific cases or we could use a query parser that tries to automatically infer the type of the query and generate one or multiple update queries based on that discovery. In either case more research on that topic is required and the lack of this feature excludes the system for a lot of applications.

6.1.2 Adding Non-Local Record

There is no possibility for the system to add new records to a data source. This means that records can only be added to the system through sources but not on a remote node via the API of the node. One reason is that there is no unambiguous mapping between a storage location for remote data and (node,source) tuple. But even if we allowed the user to explicitly specify the node and source a new data entity should be added to, it is not guaranteed that all information is available at the time of creation: if, for example, a source is backed by a database that generates record IDs automatically, we can not know that ID at the time we try to add a data item on a remote node. In addition, it is often the case that only some columns of database are mapped to fields in the type record, which leads to the question of how to fill this missing data during insertion.
6.1.3 Ambiguous Storage Location

We saw (in section 3.4.4) that storage locations for remote data are defined by setting a target location for data that matches a specific filter. Of course, it is possible that multiple filters match a single record, which would also result in multiple storage locations for a single data entity. There are multiple ways to deal with that situation, for example

- pick a random location,
- copy the data to all locations or
- choose a primary location and create filesystem link from the remaining locations to the primary.

Our implementation chooses a random location, but this may not always be the best choice. It would be desirable that the user could at least control the strategy to solve such ambiguities.

6.1.4 Auto-Commit of Changes

Changes to remote data need to be committed to the system using the API provided by the framework. In order to also have transparency for write operations, it would be nice if the system supported an optional auto-commit mode. Changes to remote data would then be detected, for example using `inotify`, and immediately committed. Only if conflicts occurred a user interaction would be required.

6.1.5 Synchronous RPC

The metadata and control channel (section 3.3.1) uses a synchronous RPC mechanism that is based on TCP connections. Tests have shown that in environments with unreliable connectivity, this leads to a lot of problems like calls that block for a very long time (until a timeout happens) and slow performance if connections are opened and closed frequently. In a productive implementation, we would suggest to exchange the RPC mechanism by a custom-made asynchronous protocol, probably based on UDP.

6.1.6 Firewalls and Address Translation

Every node listens on two subsequent TCP ports for incoming connections, one serves RPC requests and the other HTTP requests for content transmission. If nodes are connected to a network that is behind a firewall or a device that does Network Address Translation (NAT), it is required to open the ports for incoming connections and, in case of NAT, to forward the ports to the actual node. This is not very user-friendly, but there are solutions to this problem, for example Universal Plug and Play (UPnP) or UDP port knocking. Using UPnP, devices can dynamically open ports on supported NAT devices. UDP port knocking is a mechanism that opens holes in firewalls and uses them for direct communication between two firewalled hosts. On the downside, it only works for UDP, requires a non-firewalled mediator and is not guaranteed to work in all cases.
6.1.7 Delta Content Transmission
If content is not up-to-date anymore, it will be replaced by a newer version. At the moment, the whole (new) content has to be sent over the network, even if only a few bytes changed. A better approach would be to determine the differences between the versions and only transfer the changed parts. The Rsync algorithm [16] could be used to implement such a feature.

6.1.8 Security Considerations
Currently, there are no access control or encryption mechanisms implemented and every node can read all data in the system. But since both network channels are based on the HTTP protocol, they could easily be extended to use HTTP authentication or even HTTPS for encryption. Nodes could then join a group by exchanging some sort of secret during the one-time pairing phase.

Building a full-blown user management and access control is a difficult task, especially in a completely distributed system without a central instance and is beyond the scope of this work.

There is another important security consideration concerning the comparison function in the FilterElements. Currently, the comparator is passed in as Python code, which is then evaluated on all nodes. Of course, this is very insecure as potential harmful code could be distributed. In a productive implementation a dedicated language would be required to formulate filter expressions.

6.1.9 Uncontrolled Growth of Version Vectors
The synchronization version vector of a record has a relation for every node it has ever been stored on. The consequence is that version vectors always grow in size and nodes will occupy space even if they never appear again, which can become a problem if a lot of nodes have a relatively short lifespan. A solution to this problem is proposed in [8].

6.1.10 Metadata of Deleted Records
In section 2.5.5 we have seen that it is not possible to delete the metadata of a delete record completely. One approach could now be to translate the partial synchronization of records into a complete synchronization of filters. If we added an additional synchronization state (which means a set of version vectors) to the filter, that subsumes the state of all records that match to this filter, we could remove the remaining metadata of a deleted record, because this information would be implicitly given by the state of the filter. This is similar to the original vector time pair algorithm presented in Tra [12]. But additional measures would be required, because records can enter and leave filters without actually being deleted as well as records can belong to multiple filters.

6.2 Summary
We have presented a completely distributed data synchronization framework that is based on vector time pairs. It features partial replication by specifying the set of relevant data based on metadata of the content through filters, which
can in turn be either subscribed fully or partially. In both cases, the replicated data contains all the metadata of the content, but a full subscription additionally ensures that the actual content is eventually fetched. Data sources provide a flexible way to specify different sources of data and metadata, which is automatically linked into the system without requiring a separate copy of data entities within it. Data from other nodes is made available on the filesystem to the user or third-party application. This also means that no changes are required to be made to applications that want to work with data provided by the system. An API allows, among other things, to browse through all the records that are available on the node, fetch content for a specific record, commit changes to the system or to resolve synchronization conflicts.

We have successfully run the prototype implementation on desktop, notebook and Nokia N810 Internet Tablet devices, all connected through a wireless link. Even though Python does not perform very well on the tablets and the framework itself has a lot of rough edges, it shows what can be done with the system. On the other hand, the approach of attaching arbitrary sources of data through a flexible adaption mechanism seemed to be promising, it turned out that there are pitfalls which could not be solved satisfactory. We can therefore conclude that the system should be considered as a proof of concept rather than a productive application.
Bibliography


[7] Open Mobile Alliance, *OMA Data Synchronization 1.2.1*.


60


<table>
<thead>
<tr>
<th><strong>Name</strong></th>
<th><strong>Description</strong></th>
<th><strong>Output pins</strong></th>
<th><strong>Options</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Filesystem</td>
<td>Abstracts a sub-tree in a filesystem</td>
<td>_id:</td>
<td>rootPath:</td>
</tr>
<tr>
<td>DataStore</td>
<td></td>
<td>filename:</td>
<td>Recursive:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(same as _id)</td>
<td>Recursive.</td>
</tr>
<tr>
<td>SQLDatabase</td>
<td>Abstracts a generic SQL database</td>
<td>_id:</td>
<td>Query to select all data</td>
</tr>
<tr>
<td>Store</td>
<td></td>
<td>value of the column defined by idColumnName</td>
<td>UpdateQuery:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>plus: All columns returned from the select query named by column names</td>
<td>DeleteQuery:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>idColumnName:</td>
<td>idColumnName: Column name that identifies a record uniquely</td>
</tr>
<tr>
<td>MySQL</td>
<td>Abstracts a MySQL database</td>
<td>All outputs of SQLDatabase</td>
<td>All options of SQLDatabase, plus host, port, user, password, dbName</td>
</tr>
<tr>
<td>Data Store</td>
<td></td>
<td>all outputs of</td>
<td></td>
</tr>
<tr>
<td>SQLite</td>
<td>Abstracts a SQLite database</td>
<td>All outputs of SQLDatabase</td>
<td>All options of SQLDatabase, plus dbFile</td>
</tr>
<tr>
<td>Data Store</td>
<td></td>
<td>SQLDatabase</td>
<td></td>
</tr>
</tbody>
</table>
Appendix B

Adapters

<table>
<thead>
<tr>
<th>Name</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>SourceAdapter</td>
<td>0: ID of the adapted source</td>
<td>typeRecord: Type record of</td>
<td>adaptedSource: name of the</td>
</tr>
<tr>
<td></td>
<td>0: ID of the adapted source</td>
<td>the other source's record</td>
<td>source to adapt</td>
</tr>
<tr>
<td>ConstAdapter</td>
<td></td>
<td>value: Constant value (set as option)</td>
<td>value: The constant value to adapt</td>
</tr>
<tr>
<td>Filename</td>
<td>0: Filename</td>
<td>mimetype: MIME type or None</td>
<td></td>
</tr>
<tr>
<td>ToMIMETypeAdapter</td>
<td>0: Filename</td>
<td>file: Root path of the</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>extension: Extension part of</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>the filename</td>
<td></td>
</tr>
<tr>
<td>SplitFilenameAdapter</td>
<td>0: Filename</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIMETypeToFileExtensionAdapter</td>
<td>0: MIME type</td>
<td>extension: File extension that matches the MIME type</td>
<td></td>
</tr>
<tr>
<td>StringLengthAdapter</td>
<td>0: A string</td>
<td>length: Length of the string</td>
<td></td>
</tr>
<tr>
<td>MD5HashAdapter</td>
<td>0..n Variable amount of file objects or strings</td>
<td>hash: MD5 hash of the concatenated input</td>
<td></td>
</tr>
<tr>
<td>ConcatenateAdapter</td>
<td>0..n An arbitrary amount of strings</td>
<td>concatenation: Concatenation of the input strings</td>
<td></td>
</tr>
</tbody>
</table>
## APPENDIX B. ADAPTERS

<table>
<thead>
<tr>
<th>Name</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileStat Adapter</td>
<td>0: filename</td>
<td>file:</td>
<td>File supplied as input</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>path:</em> Path part of the file</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>name:</em> Filename only part</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>size:</em> Size in bytes of the file content</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>atime:</em> Time of last access</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>mtime:</em> Time of last modification</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>ctime:</em> Time of creation</td>
</tr>
<tr>
<td>FileOpen Adapter</td>
<td>0: filename</td>
<td>fileobject:</td>
<td>File-like object wrapping the file</td>
</tr>
<tr>
<td>BLOBAdapter</td>
<td>0: String</td>
<td>fileobject:</td>
<td>File-like object wrapping the BLOB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>size:</em> Size in bytes of the BLOB</td>
</tr>
<tr>
<td>Content MetaData</td>
<td>0: A file-like object</td>
<td>Outputs depend</td>
<td>on the type of the file being adapter</td>
</tr>
<tr>
<td>Adapter</td>
<td>to extract</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>metadata from</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix C

RecordLock

def acquireReadLock(self, recordUUID):
    with self._lockCV:
        while recordUUID in self._writeLocks:
            self._lockCV.wait()
        if self._readLocks.has_key(recordUUID):
            self._readLocks[recordUUID] = self._readLocks[recordUUID] + 1
        else:
            self._readLocks[recordUUID] = 1
    return

Listing C.1: Acquiring read locks

def releaseReadLock(self, recordUUID):
    with self._lockCV:
        rlcoun = self._readLocks[recordUUID]
        if rlcoun == 0:
            raise Exception("Tried to release unlocked record \(\%s\) \% (recordUUID))
        elif rlcoun > 1:
            self._writeLocks[recordUUID] = rlcoun - 1
        else:
            self._readLocks.pop(recordUUID)
        self._lockCV.notifyAll()
    return

Listing C.2: Releasing read locks

def acquireWriteLock(self, recordUUID):
    with self._lockCV:
        while recordUUID in self._writeLocks:
            self._lockCV.wait()
        self._writeLocks.add(recordUUID)
        while self._writeLocks.has_key(recordUUID):
            self._lockCV.wait()
    return

Listing C.3: Acquiring write locks

def releaseWriteLock(self, recordUUID):
    with self._lockCV:
        if not recordUUID in self._writeLocks:
            raise Exception("Tried to release unlocked record \(\%s\) \% (recordUUID))
        self._writeLocks.discard(recordUUID)
        self._lockCV.notifyAll()
    return

Listing C.4: Releasing write locks