*Shekow, M.; Prinz, W. (2019): A capability analysis of groupware, cloud and desktop file systems for file synchronization. In: Proceedings of the 17th European Conference on Computer-Supported Cooperative Work: The International Venue on Practice-centred Computing an the Design of Cooperation Technologies - Exploratory Papers, Reports of the European Society for Socially Embedded Technologies (ISSN 2510-2591), DOI: 10.18420/ecscw2019_ep06*

# A capability analysis of groupware, cloud and desktop file systems for file synchronization

Marius Shekow and Wolfgang Prinz
Fraunhofer FIT, Sankt Augustin, Germany
*Contact Author: marius.shekow|wolfgang.prinz@fit.fraunhofer.de*

**Abstract.** Many groupware applications use hierarchical file systems, cloud storage or shared desktop operating system disks to support the cooperative development of shared artefacts or to share information. In these collaboration scenarios, often file synchronizers assist users in the data management across multiple devices. They establish consistency between file systems, even in light of their heterogeneity. However, the development of file synchronizers is difficult due to the fact that mainstream operating systems were not primarily built for cooperation or synchronization scenarios. Further, synchronizers need to address heterogeneity, by translating semantical differences and considering cross-device and cross-file system incompatibilities. This paper provides an in-depth analysis of six file system capabilities relevant to shared data synchronizers, such as mapping from namespace to physically stored objects, supported object types, namespace limitations or locking mechanisms. For each capability we derive commonalities for a set of selected file systems and also provide advice for handling incompatibilities. The insights of this work provide useful concepts and guidance for groupware developers that aim for a better user experience in synchronization support.

## 1 Introduction

With the increasing availability, affordability and mobility of computing devices like laptop computers, smartphones and tablets, working with multiple devices in both professional and private life has become increasingly common. Users typically use applications like word processors or other domain-specific tools to create large parts of their data. The resulting documents are stored on the devices in a hierarchical *file system*, which has the role of a persistent database. Many different collaboration scenarios exist for files, such as users working together on office documents or file-based databases. To facilitate collaboration and to increase availability, documents are exchanged, e.g. via e-mail or via central systems, such as groupware, file servers or ubiquitous cloud storage. However, copying files and directories between storages causes problems, both for an individual user who wants to manage her files across these storage systems [1] and for collaborative multi-user scenarios.

One convenient solution is *data synchronization*, which has become popular, also due to the increased availability and affordability of cloud services (Yang et al., 2016). *File synchronizers*, as described by Balasubramaniam and Pierce (1998), are synchronizers whose data is the file system, including its namespace structure and file contents. In particular, cloud storage-based file synchronizers like Dropbox, Google Backup and Sync, OneDrive or NextCloud have become popular over the last ten years, indicated by the high number of their users (Kollmar, 2016; Price, 2017). They are programs that constantly run on a device in the background and tightly integrate with the file manager, providing a native user experience. They eliminate friction in file-based workflows because users no longer need to use 3rd party systems (such as a cloud storage web interface) but can work on the local file system directly, which avoids manual up- and downloads which cause files to lose their context (Vonrueden and Prinz, 2007). The offline availability of files improves navigation and search speed in the file system hierarchy. The native integration of a cloud synchronizer into the operating system and file manager provides many advantages. It makes 3rd-party functionality available at the user's finger-tips, such as the file manager's context menu which provides direct access to previous versions or comments of a file. Synchronizers also provide *synchronous awareness* (Fuchs et al., 1995), e.g. by showing native notifications in case new files were created, opened or locked by other users.

Today a plethora of industrial file synchronizers have emerged[2], used by a large user base. They need to support the file system APIs of all end-user operating systems they run on, as well as the API of the central file system. Building such file synchronizers is challenging, for several reasons. This work focuses on the fact that no two file systems are exactly equal, due to their heterogeneous capabilities. We use the term *capability* for a specific characteristic of a file system, such as namespace limitations or the way object relationships are modeled. Their traits may be different (heterogeneous) between any two file systems. If the

[1] Exemplary, users may fail to locate the correct, up to date version of a document on the right device. See e.g. Dearman and Pierce (2008); Jokela et al. (2015) for more details.

[2] E.g. Dropbox, Google Drive, Microsoft OneDrive, Amazon Drive, Box, NextCloud, Cloudstore, Resilio, Seafile, SpiderOakOne, LeitzCloud, Tonido, TeamDrive, MyDrive, Strato HiDrive, or Hubic. See (Wikipedia, 2017a) for a more complete list.

synchronizer developer ignores or overlooks a capability, this impairs the usability of the system because of bad side effects that occur during synchronization. Exemplary, if a developer overlooks that a file may not be named "aux" on Windows, the Windows implementation will run into unexpected loops or errors while trying to synchronize such a file, which was synchronized successfully by the macOS implementation. We have observed several instances of such side effects in practice in leading industrial synchronizers. The result is either "just" a divergence of the file systems, or worse, data loss.

We created this work as part of an ongoing endeavor to build a file synchronizer that overcomes the shortcomings of existing solutions supporting multi-user collaboration in asynchronous cooperation scenarios. We identify both homogeneous and heterogeneous capabilities relevant to file synchronizers. We propose suitable data transformation, where applicable, to avoid data loss. We start in section 2 where we briefly explain the mechanics of a file synchronizer and examine the variety of ways how file synchronizers define their file system. Next, we introduce five representative file systems we examined in section 3. In section 4 we present the detailed analysis of six capabilities. We conclude and present future work in section 5.

## 2 Background

Although the synchronization of information is essential for the support of collaborative work the CSCW research community focused primarily on researching synchronous synchronization and consistency algorithms such as the seminal work of Ellis and Gibbs (1989) on operation transformation and subsequent research by Sun and Ellis (1998); Sun and Sun (2009). On the other hand the CSCW community indicated the importance of consistent and contextual information sharing process (Voida et al., 2006). Although relevant for CSCW, file synchronization has primarily been researched in other domains. The seminal work by Balasubramaniam and Pierce (1998) describes and coins the term *file synchronizer*. We address authors of similar (or more powerful) synchronizers. A file synchronizer is a program that performs a pair-wise synchronization of two file system replicas upon the user's request, breaking synchronization down to a 3-stage process. In the first stage, update detection, the local and remote replicas are scanned to detect their current state. The list of changes (updates) is computed by comparing the current state to a locally persisted state from the point of the last synchronization. The second stage, reconciliation, is given the updates of both replicas and simulates (in memory) how the final, reconciled file system should look like which contains the updates of both replicas. The updates are examined for conflicts for which the user is asked to choose a suitable resolution. The output of this stage is the list of operations for the user to review in a graphical interface. The final stage, propagation, performs the actual file system modifications on each replica and updates the locally persisted state.

Such file synchronizers convert the file system from being a *collaboration-transparent*, replicated architecture to a *collaboration-aware* system (Phillips, 1999). Due to the heterogeneous capabilities of file systems, such synchronizers are also referred to as heterogeneous (Antkiewicz and Czarnecki, 2008; Foster et al., 2007). The advantage of heterogeneous synchronizers is that users can continue using existing file systems, without the (expensive) migration to a homogeneous system. The disadvantage is that the developer needs to build an internal model that is as compatible as possible with every file system the synchronizer aims to support. This involves finding a set of *common* capabilities, which we are doing in this work. Typically, the synchronizer *transforms* the heterogeneous model of each file system to the internal one. The transformation is challenging, because a suitable alignment needs to be found. The synchronizer then decides which updates to synchronize using the internal model. In extreme cases parts of the data are lost due to lack of alignment, as our work will show.

While there is a large number of industrial file synchronizers, the body of academic works is much smaller. We examined whether related works define a formal and thorough specification of their file system model, because we consider a formal definition of the data schema and its rules a basic requirement for any data synchronizer. Interestingly, a few works do not provide any specification of the file system and its operations, see e.g. (Cox and Josephson, 2005; Elijorde et al., 2013). Some provide a partial description, such as the record structure used to store the file system's state or the operations, see (Lindholm et al., 2005; Molli et al., 2003; Tao et al., 2015; Bao et al., 2011; Li et al., 2012; Uppoor et al., 2010). Others such as (Balasubramaniam and Pierce, 1998; Ng and Sun, 2016; Ramsey and Csirmaz, 2001; Csirmaz, 2016) formally specify a file system they *defined*. These works do not discuss the *mismatch* that exists between their internal model and the real-world file system their implementation actually works on.

Real-world file systems specifications, such as POSIX, are only formulated informally. A few academic works such as Ridge et al. (2015) exist which extracted exhaustive first-order logic (FOL) specifications for a few real-world implementations, but not all main stream file systems are covered yet. We present an *informal* comparison in this work instead, as this allows the provision of immediate results for a large selection of file systems. Some online resources such as (Craighead, 2008; Wikipedia, 2017b) also provide informal comparisons. Apart from (Jim et al., 2002), an unfinished manuscript by the authors of (Balasubramaniam and Pierce, 1998), there is no related scientific literature to the best of our knowledge that provides an in-depth discussion of the capabilities of file systems.

## 3 Examined file systems

To find capabilities we sample different *types* of file systems. As selection criteria we focus on market share and system type and chose one or two representative systems for each type. We examine Windows version 7-10 (NTFS) and macOS version 10.11-10.13 (HFS+ and APFS) APIs because these are the most widespread end-user operating systems at the time of writing. Our findings also transfer to UNIX and therefore to both file servers (e.g. network-attached storage) and mobile devices such as smartphones. We consider WebDAV (Dusseault, 2007) which is widely available as interface for proprietary as well as open-source Internet (cloud) storages. Dropbox (HTTP API v2 (Dropbox Inc., 2017)) is chosen as a representative for widespread cloud storages (Dropbox Inc., 2016). *BSCW Social* (OrbiTeam Software GmbH & Co KG, 2018) is a representative for groupware systems commonly found in academia, a system that originates from the CSCW community (Bentley et al., 1997; Jeners and Prinz, 2014).

## 4 Capability analysis

This section provides an in-depth analysis of six capabilities relevant to file synchronizers. They were selected based on technical realities we discovered while implementing and technically evaluating a file synchronizer. Each capability is discussed in a separate subsection. For each one we first state its significance for the user, followed by an analysis, then extract similarities that manifest in the file synchronizer's internal model and finally give advice how file synchronizers can handle incompatibilities, if applicable.

### 4.1 Physical object & namespace mapping

The *namespace* is the user-facing side of a file system. It consists of a hierarchical set of *paths*, where a path is a notation for addressing a specific *object*. A path is a sequence of *names*, where names are simple strings. Hierarchy levels of a path are separated by a *separation character*, such as '/' or '\'. File system implementations differ in their approach how objects are identified, physically stored and how the mapping between namespace and objects works.

#### 4.1.1 Significance

From the user's perspective the synchronizer translates a prefix of the synchronized namespace between the local disk and the remote storage, e.g. *'C:\SyncFolder'* to *'https://server.com/synced'*. Users expect that the local disk's and the server's namespace match exactly. However, due to technical limitations (analyzed below) this is not always possible. A synchronizer that is aware of incompatibilities should find a suitable way to inform the user about namespace mismatches (Dourish, 1996).

#### 4.1.2 Analysis

An overview of the analysis is shown in figure 1.

We first classify whether file system objects (files, directories, etc.) can be identified uniquely (e.g. after moving them) by a persistent identity, or whether

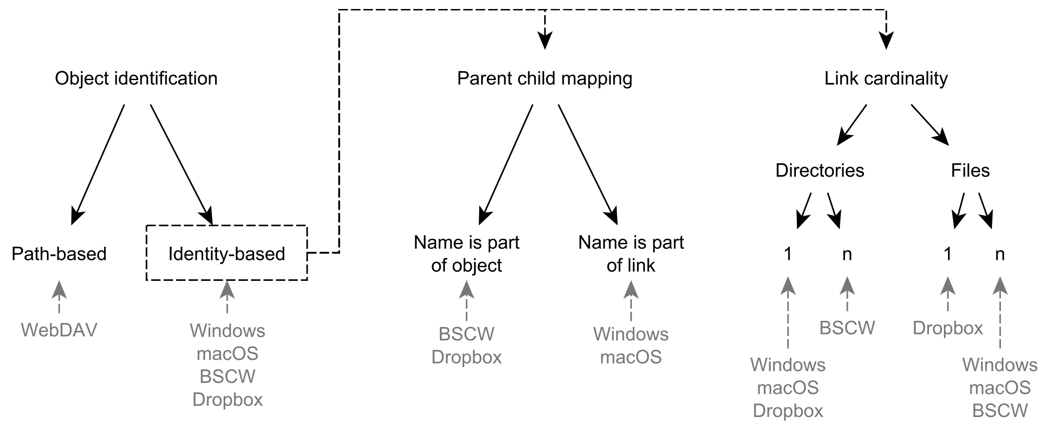

Figure 1. Analysis of object identification and namespace to object mappings.

only the path is available (Tao et al., 2015). Exemplary, Windows provides the *file index*, and macOS or UNIX systems provide *inode* numbers. For identity-based systems, two further classifications are appropriate, because an object with a specific ID may be accessible from one or more paths. In practice the cardinality varies per object type, s.t. Windows or macOS forbid more than one link to a directory to prevent cycles to occur in the tree. Some systems model the parent child relationship s.t. each directory has a list of $(name, id)$ tuples of its immediate children (name of the objects is part of the *link*), whereas others store the name as part of the object and each directory maintains a simple list of immediate child IDs.

Two more aspects not covered in figure 1 are that the invariants of each file system need further examination. A file system may or may not allow two sibling objects to have the same name, and it may use a case-sensitive or case-insensitive comparison while enforcing this invariant.

#### 4.1.3 Derived unified model

To derive the internal file system model we suggest the following approach:

- If one or more file systems are *path-based*, either let the internal model be path-based too, or emulate IDs by generating them on the client, setting IDs as custom meta-data, if the file system API supports it (e.g. WebDAV PROPPATCH, see section 9.2 of Dusseault (2007)).
- When the parent child mapping varies, let the name be part of the object.
- If link cardinality varies, use the smaller (1) cardinality.
- When invariants vary, enforce the one that is most strict.

#### 4.1.4 Advice for handling incompatibilities

When a file synchronizer encounters an incompatible mapping at run-time, e.g. if a specific file exists at multiple paths but the internal model limits the cardinality to 1, we suggest the synchronizer either stops synchronizing, asking the user to fix the situation, or to automatically add the affected paths or IDs to an *ignore list*. Numerous industrial synchronizers provide such an ignore list users can fill with paths to files or directories they want to exclude from synchronization. We suggest that this list can also be manipulated by the reconciliation algorithm automatically to handle compatibility issues, notifying the user in such an event. For certain traits, workarounds may be possible. Exemplary, junctions (Windows) and symbolic links (macOS) may be used to allow a $N$-cardinality for directories. The synchronizer needs to choose one path as primary and use junctions or links for all other paths, updating them in case the primary path changes.

## 4.2 Supported object types

Files and directories are the two object types offered by all examined file systems. Jeners et al. (2013) show that even in groupware systems such as BSCW which offer many additional object types, the majority (90%) of user interaction takes place with these two object types. A file system may also support other object types that are incompatible with other systems.

### 4.2.1 Significance

When an object available on one file system is unavailable on the other one, its omission in the namespace, which is a loss of information, will confuse the user.

### 4.2.2 Analysis

While all examined file systems offer files and directories, there are several other types supported by just a subset of file systems, e.g. device files or symbolic links on macOS and Windows, or special types like contact lists, calendars or URLs on BSCW.

### 4.2.3 Derived unified model

By taking the intersection set of the available object types of each file system, the internal model should consist only of files and directories. We suggest to ignore other object types because they are specific to that file system and cannot be meaningfully viewed or manipulated on other systems that do not support them.

### 4.2.4 Advice for handling incompatibilities

We propose a similar handling as for mapping issues (section 4.1) where the synchronizer either stops or adds affected objects to the ignore list automatically, notifying the user about this action. A workaround is to create proxy objects, such as '.url' files, that allow the user to see the existence of the corresponding objects, redirecting the user to the respective location on the other file system in case she opens the proxy object.

## 4.3 Operations and atomicity

File system APIs offer many operations to both *query* the current state of the file system (e.g. listing a directory's content) or to *manipulate* it. In the update detection stage a file synchronizer relies on the query operations to extract the current state. At the final *propagation* stage, the synchronizer needs to transform the scheduled abstract operations (which equalize both file systems) to concrete operations of each file system. This is challenging because the exact operations, their preconditions and their degree of atomicity[3] vary.

### 4.3.1 Significance

A user expects that operations she applied to her local file system are consistently applied to other file systems by the synchronizer. Users also expect the synchronizer to avoid inconsistent states while synchronization is active or was interrupted. Not handling related issues causes confusion (e.g. attempting to open a partially transferred file) or additional work (such as manually cleaning up inconsistent files and directory structures) for the user.

### 4.3.2 Analysis

Every of the examined file systems offer operations to query the current state. The slight variations in query operation signatures are merely an implementation detail. When considering manipulation operations, all file systems offer operations to create or delete empty directories, or to move an object. However, there is significant variation in the availability and atomicity of operations used to create or update files, or to delete non-empty directories. Exemplary, BSCW allows to atomically create non-empty files or delete non-empty directories, while Windows does not. Another observation is that desktop file systems like Windows and macOS offer *mount* operations which create a mount point that establishes a transition between volumes.

### 4.3.3 Derived unified model

A user would expect a file synchronizer to be capable of a set of operations the user also knows from using the file manager. An exemplary list could be as follows:

- *createdir(path)* creates an empty directory at *path*
- *deletefile(path)* deletes the file at *path*
- *deletedir(path)* deletes the directory and all its children at *path*
- *move(source, dest)* moves an existing object from *source* to *dest*
- *transfer(source, dest)* transmits a *file* located at *source* on the source file system to *dest* on the destination file system, to create a new file or update an existing one

[3] We refer to *atomicity* as known from database systems, see also section 1.3.4 of Elmasri and Navathe (2015).

To not leave either file system in an inconsistent state, every operation is expected to succeed or fail atomically. Optionally, a *copy file* operation can be used to copy a file on the destination file system in case it is feasible to detect exact copies of files on the source file system, e.g. by using checksums.

#### 4.3.4 Advice for handling incompatibilities

All discrepancies we found between concrete file system operations and the ones presented above result from varying degrees of atomicity, which can be solved in the following ways:

- *deletedir(path)*: if a file system does not offer an atomic, recursive implementation, we suggest to first call *move(path, temp)* where *temp* is a path outside of the synchronized namespace, but on the same volume. This move operation succeeds (or fails) atomically and *appears* as an atomic delete operation to the synchronizer. Next, perform a *post-order* traversal of *temp*'s sub-namespace, deleting first files then directories.
- *transfer(source, dest)*: if the destination file system's operation is not atomic, we propose to execute *transfer(source, temp)*, i.e., write transferred data to a temporary location *temp* that is outside the synchronized namespace but also on the same volume. Once finished, perform *move(temp, dest)* on the destination file system.

Finally, file synchronizers which detect move operations via the object's ID should be aware of *mount points* within the synchronized namespace. IDs are only unique within a volume. However, a mount point establishes a transition between volumes. When the user performs a conceptual *move(source, dest)* operation where *source* is on volume *A* and *dest* on volume *B*, the synchronizer will incorrectly detect a *delete* operation for *source* and a *create* operation for *dest*. We therefore suggest that synchronizers detect mount points and either reject them (by stopping synchronization) or automatically adding them to the *ignore list*.

### 4.4 Namespace limitations

Although the general namespace consists of *Unicode* characters, a file system may pose limitations on the namespace, affecting paths or the names of a path, usually for technical or historical reasons.

#### 4.4.1 Significance

When a user attempts to create an object with a name that violates a namespace limitation, the file manager (or web interface) prevents the creation and provides *immediate* feedback how to fix the name. When using file synchronization, the chosen name may be accepted by the source file system API, but may violate a limitation of the destination API. The file synchronizer discovers this issue after a (possibly large) delay which surprises the user, because to her the creation of the object initially appeared to be successful. Furthermore, users will be confused if objects exist on one system but not the other one due to a limitation that affects only the latter system.

### 4.4.2 Analysis

The following list provides a brief summary of our findings. We refer to the respective file system documentation for further details[4].

- A file system may reserve a set of *characters* from being used in object names, either at any position, or only in specific positions. *Forward slashes* are forbidden in all examined systems, as they separate names in a path. Windows reserves the most characters, and other systems such as BSCW or Dropbox have adopted Windows' set of reserved characters and names for compatibility reasons.
- Similarly, some systems reserve a set of *names*, such as "." or "..". Windows reserves a large set of names such as "CON" or "PRN" for historical reasons and also reserves *short file names* (Microsoft Inc., 2018) in case a longer file name already exists (exemplary, given a directory named "project report", creating an object at "projec~1" is forbidden on volumes with short file name creation enabled).
- Many systems impose a maximum length of names and paths. Often names are limited to a length of 255 characters. Shorter path lengths (such as macOS with 1016 characters) also cause issues, e.g. deep directory hierarchies being in accessible.
- While all examined systems use the Unicode alphabet with some form of encoding (e.g. UTF-8), not all systems *preserve* the *normalization form* (such as NFC or NFD[5]) of characters. Exemplary, the HFS+ file system on macOS does not preserve a large set of input characters but converts them to a NFD-like form.
- *Case-sensitivity* may vary between two file systems. By default, the *examined* systems are all case-insensitive. However, others such as the UNIX file system, are case-sensitive! We found all systems to be case-*preserving*.
- In rare instances the file system APIs behave deceptively. They accept a name, seemingly execute successfully, but actually change the name internally. This is problematic for file synchronizers, as the next update detection phase will find an unexpected name and assume that the object was moved by the user. One example is the Unicode normalization conversion of HFS+ volumes mentioned above, another is Windows which silently strips trailing spaces/dots from a name during execution.

[4] See e.g. Berners-Lee et al. (1994), Apple Inc. (2004), Apple Inc. (2017) or Microsoft Inc. (2018).

[5] See *http://unicode.org/reports/tr15/*, retrieved January 2, 2019.

#### 4.4.3 Derived unified model

For each limitation of file systems *A* and *B* we propose to take the one that is more strict and let the file synchronizer apply it to the file system with the weaker limitation. For reserved characters or names this means to apply the *union* of the sets to both *A* and *B*. For length limitations, the shorter length is more strict. Also, case-insensitivity is more strict than case-sensitivity.

#### 4.4.4 Advice for handling incompatibilities

We suggest a file synchronizer takes one of the following approaches when encountering paths that are incompatible w.r.t. the unified limitations:

1. Stop synchronization, ask the user to manually rename objects
2. Automatically rename objects to establish compatibility
3. Automatically add incompatible objects to the ignore list

While approach (1) is easy to implement, it is labor-intensive for the user. In case the stopped synchronization goes unnoticed, and if it remains in that state for extended periods of time, this increases the chance for conflicts. Approach (2) mitigates this problem, but automatic renaming can cause issues when the affected objects belong to a naming scheme of a third party application. Such applications may stop working once these files and directories no longer correspond to the expected naming scheme. The last approach fixes the issues of the two ones but requires the implementation of the aforementioned ignore list.

### 4.5 Meta-data

Meta-data provides further information about objects. It is not stored as part of the object, but at a separate location.

#### 4.5.1 Significance

When meta-data stored on one file system is incompatible with the other file system, a synchronizer must skip their synchronization or perform a conversion. This type of data loss negatively affects the user, because she cannot access meta-data available only on the remote file system during an offline period.

#### 4.5.2 Analysis

Each file system provides a diverse set of meta-data. Some meta-data are *attributes* managed by the file system, others can be changed by a client applications, such as a file synchronizer. Some systems offer one or more APIs to write *custom* meta-data, e.g. *Extended Attributes* and *Alternate Data Streams* on Windows, or *xattr* and *Resource forks* on macOS. The following meta-data is available on *all* file systems:

- Object type (file, directory, ...)
- File size (for files)
- Timestamp of creation and last modification

#### 4.5.3 Derived unified model

All file systems support the retrieval of meta-data that is necessary to extract their state, such as the object's type or the last-modified timestamp. In case a file synchronizer models the file system using IDs, all file systems except for WebDAV automatically generate and provide unique IDs. For WebDAV we propose that the *file synchronizer* generates globally unique IDs (GUIDs) when creating objects on a WebDAV file system, assigning the GUID via the *PROPPATCH* command.

#### 4.5.4 Advice for handling incompatibilities

Some meta-data, such as *attributes*, are system-specific and often lose meaning when copied to another file system, especially when it is of different type or located on a different operating system or machine. Exemplary, synchronizing the *compressed* attribute of a Windows file to the corresponding file on a macOS file system defies any purpose. We find that bypassing meta-data synchronization largely facilitates a file synchronizer's implementation. This also applies to *authorization* mechanisms, such as UNIX *permissions* or the more powerful *Access Control List* entries, which can also be considered to be meta-data, with varying availability and heterogeneity.[6]

The *last-modified* timestamp is an exception. We suggest to synchronize it because it is typically available on each file system, has the same meaning everywhere and users are aware of it when using the file manager. A caveat developers need to consider is the variety of resolutions and formats of timestamps.

## 4.6 Locking

*Locking* allows one user to *exclusively* modify an object on a file system, while all other users are prevented from modifying their own replica of that object.

#### 4.6.1 Significance

Locking is an important mechanism that introduces *pessimistic* concurrency control in situations where users expect that conflicts are likely to happen. It avoids conflicts or lost updates. In an example scenario, a user locks a document she exclusively wants to work on for an hour. During this time, other users should be unable to concurrently modify this file, and should be *aware* of this lock while it is set. The

[6] As an example for heterogeneity, macOS and Windows both support *Access Control Lists*, but their implementations vary considerably. Additionally, synchronization of authorization data would require to also synchronize *authentication* data, i.e., user accounts, which introduces additional challenges.

information about the lock's existence can be propagated by the synchronizer to other users while they are online. In practice we have not observed locking to play a role for files stored on *local* disks. However, this feature is frequently used in groupware systems such as BSCW, and the transparent handling and awareness of locking behavior is an early requirement for CSCW systems as described in Blair and Rodden (1994).

### 4.6.2 Analysis

We analyzed the file systems' locking capabilities to determine whether a file synchronizer can safely protect an object from modification by the local user, because a different user locked the object. We found that some systems such as Dropbox do not offer any locking mechanism. Systems such as WebDAV and BSCW provide an elaborate locking model, including lock meta-data such as the owner and expiration time.

The locking mechanisms of Windows (*read-only* attribute, *file handle* locking) and macOS (*immutable* attribute, advisory locks via *fcntl*[7] API) are less elaborate. They each work differently and protect other aspects of modification. Exemplary, the read-only attribute on Windows does not protect objects from being moved or renamed, while the immutable attribute on macOS does.

We think that this diversity stems from the fact that each mechanism has a different purpose. On Windows and macOS the read-only/immutable file attribute or handle-based locks were not designed for a multi-user locking scenario. It is our understanding that they exist to allow users (and programs) to protect objects from modification *on the same device*, not across multiple devices. Handle-based locking suffers from *volatile* characteristics[8]. On macOS, handle-based locking is designed for a set of *cooperating* programs and not intended to prevent third party programs from modifying files. On Windows, handle-based locking has more wide-spread effect than just locking the object itself. It works on a "first come, first served" basis. Even just opening a file for reading already locks it. A file synchronizer may fail to obtain a lock, or inadvertently lock the path of any parent object, which is not desired. In addition, reliable recursive locking of a *directory* is not possible with the mechanisms offered by Windows and macOS.

### 4.6.3 Derived unified model

In case a pair-wise synchronization targets two file systems of *equal* type, such as two WebDAV systems, lock synchronization is feasible. In any other scenario we advise to ignore lock synchronization due to the strong differences in their implementation, making it impossible to meaningfully map one lock type onto another one.

[7] *http://man7.org/linux/man-pages/man2/fcntl.2.html*, retrieved January 2, 2019.

[8] When the program that owns the handle to an object terminates, the lock is automatically cleared.

### 4.6.4 Advice for handling incompatibilities

Not synchronizing locks does not necessarily mean that the synchronizer completely ignores locks. Some systems like WebDAV allow the *discovery* of locks (*before* the attempt of modifying a locked resource). Assume a scenario where a synchronizer detects that the user updated file $f$ locally, while $f$ is locked on the remote replica by another user. The synchronizer may then skip synchronizing $f$ and notify the user about the lock's existence. With additional implementation effort, a synchronizer may also monitor the user's opened files and warn her in case she opens a file that is locked by other users. It is also possible to convey the existence of locks by the use of *overlay icons* in the file manager.

If lock discovery is unavailable we propose to treat failures like any other permission-related ones, such as failures resulting from prohibitive ACL entries or UNIX permissions. The synchronization may be stopped or the affected object could be skipped. The user should be notified about the problem in either case and be provided with as much available information as possible to fix the problem.

## 4.7 Summary

A summary of the capabilities of each file system is shown in figure 2. This radar chart depicts a rough estimate of the degree of power for each capability from 0% (center) to 100%, based on a technical evaluation beyond the scope of this work. Smaller values indicate less powerful namespace mappings, fewer supported object types, stronger namespace limitations, smaller level of locking, etc. We chose 20% as minimum value only to improve readability. By intersecting the areas of the file systems a synchronizer supports we can derive the degree of limitations of the synchronizer's internal model.

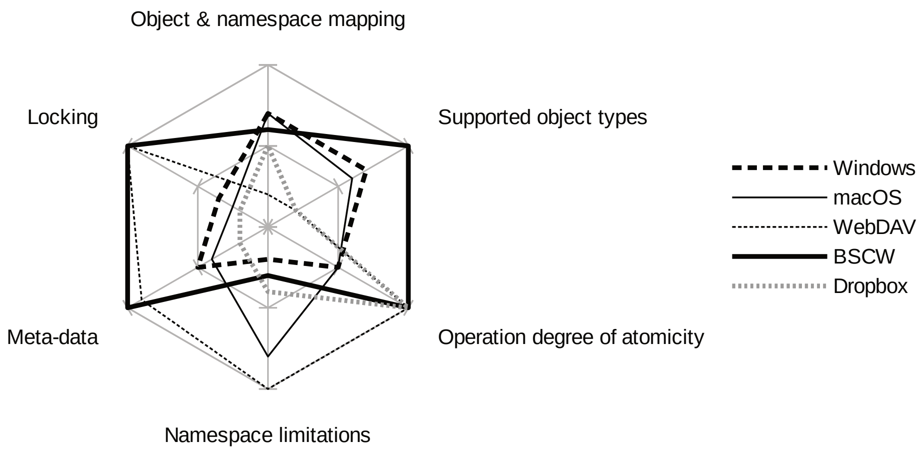

Figure 2. File system capabilities overview.

## 5 Conclusions and future work

In this work we have analyzed several file system capabilities relevant to file synchronizers as a baseline for the development of cooperation support applications. Synchronizers facilitate data management and collaboration in single- and multi-user settings. Supporting a variety of heterogeneous systems satisfies the user's need to synchronize between different devices and services, thus aiming at the provision of an integrated collaboration environment (Prinz et al., 2009).

Synchronization of the examined file systems is challenging due to their heterogeneity. This first and foremost affects the structure of a file system. As we discussed in section 4.1 and 4.2 two file systems may vary how paths of the namespace are mapped to objects or which types of objects exist. We proposed to generally take the lowest common denominator, e.g. limit synchronization to files and directories, or allow each object to be linked just once into the namespace. We proposed that the most user-friendly solution to deal with incompatible paths is to add them to an *ignore list* automatically. In subsection 4.3 we informally presented a set of commonly available operations that are sufficient to achieve consistency. Some of them require a degree of *atomicity* not offered by some implementations like Windows and macOS. For these we provided workarounds which emulate atomic behavior. In section 4.4 we found that Windows is imposing strong namespace limitations due to a large set of reserved names and characters. BSCW and Dropbox mimic Windows' behavior for compatibility reasons. Consequently, objects with incompatible names need to be dealt with, for which we presented several approaches, each with their own advantages and disadvantages. We analyzed accessible meta-data in section 4.5. Except for WebDAV, all file systems provide an automatically generated object ID which allows a file synchronizer to uniquely identify objects irrespective of their path, which facilitates the detection of move operations. Except for the *last-modified* timestamp we consider synchronization of other meta-data inadequate. Finally, section 4.6 discusses locking. We find that lock semantics of two file systems of different type are too heterogeneous to allow for a meaningful lock synchronization. Where possible, synchronizers should provide *awareness* of active locks to the user.

Despite the discussed caveats we still consider the use of file synchronization an enrichment of the user's experience. We hope that authors and developers of file synchronizers find our in-depth analysis and advice useful when implementing heterogeneous file synchronizers. While a lot of the given advice for handling incompatibilities may appear straightforward, our analysis of several industrial file synchronizers has shown a great variety in behavior, including many illogical choices[9]. We also hope that developers of next-generation file systems may also find clues to build systems that better support synchronization, in particular considering aspects such as *locking* in a cooperative setting. As future work we

[9] Exemplary, the macOS implementation of OneDrive uploads files with Windows-reserved *names* without warning, but skips synchronization of reserved *characters*.

will analyze the effects of how a file synchronizer models the state and operations of a file system on the conflicts that it detects, including a discussion of conflict resolution approaches taken by different related works. A user study is planned to verify our recommendations of handling incompatibilities with users.

## References

Antkiewicz, M. and K. Czarnecki (2008): ‘Design Space of Heterogeneous Synchronization’. In: R. Lämmel, J. Visser, and J. Saraiva (eds.): *Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 3–46.

Apple Inc. (2004): ‘Technical Note TN1150: HFS Plus Volume Format’. https://developer.apple.com/legacy/library/technotes/tn/tn1150.html.

Apple Inc. (2017): ‘Apple File System Guide - FAQs’. https://developer.apple.com/library/content/documentation/FileManagement/Conceptual/APFS_Guide/FAQ/FAQ.html.

Balasubramaniam, S. and B. C. Pierce (1998): ‘What is a File Synchronizer?’. In: *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking*. New York, NY, USA, pp. 98–108, ACM.

Bao, X., N. Xiao, W. Shi, F. Liu, H. Mao, and H. Zhang (eds.) (2011): ‘SyncViews: Toward Consistent User Views in Cloud-Based File Synchronization Services: 2011 Sixth Annual Chinagrid Conference’.

Bentley, R., T. Horstmann, and J. Trevor (1997): ‘The World Wide Web as Enabling Technology for CSCW: The Case of BSCW’. *Computer Supported Cooperative Work (CSCW)*, vol. 6, no. 2, pp. 111–134.

Berners-Lee, T., L. Masinter, and M. McCahill (1994): ‘Uniform Resource Locators (URL)’.

Blair, G. S. and T. Rodden (1994): ‘The Challenges of CSCW for Open Distributed Processing’. In: *Proceedings of the IFIP TC6/WG6.1 International Conference on Open Distributed Processing II*. Amsterdam, The Netherlands, The Netherlands, pp. 127–140, North-Holland Publishing Co.

Cox, R. and W. Josephson (2005): ‘File Synchronization with Vector Time Pairs’.

Craighead, M. (2008): ‘Windows vs. Unix File System Semantics’. http://www.conifersystems.com/2008/10/21/windows-vs-unix-file-system-semantics/.

Csirmaz, E. (2016): ‘Algebraic File Synchronization: Adequacy and Completeness’. https://arxiv.org/pdf/1601.01736.pdf.

Dearman, D. and J. S. Pierce (2008): ‘It’s on my other computer! Computing with multiple devices’. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Florence, Italy: ACM, pp. 767–776.

Dourish, J. P. (1996): ‘Open implementation and flexibility in CSCW toolkits’. phd, University of London, London.

Dropbox Inc. (2016): ‘Celebrating half a billion users’. https://blogs.dropbox.com/dropbox/2016/03/500-million/.

Dropbox Inc. (2017): ‘Dropbox for HTTP Developers: The Dropbox API v2’. https://www.dropbox.com/developers/documentation/http/overview.

Dusseault, L. (2007): ‘HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)’.

Elijorde, F. I., H. Yang, and J. Lee (2013): ‘Ubiquitous Workspace Synchronization in a Cloud-based Framework’. *Journal of Korean Society for Internet Information*, vol. 14, no. 1, pp. 53–62.

Ellis, C. A. and S. J. Gibbs (1989): ‘Concurrency control in groupware systems’. In: *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*. Portland, Oregon, USA: ACM, pp. 399–407.

Elmasri, R. and S. B. Navathe (2015): *Fundamentals of database systems*. Pearson.

Foster, J. N., M. B. Greenwald, C. Kirkegaard, B. C. Pierce, and A. Schmitt (2007): ‘Exploiting schemas in data synchronization’. *Journal of Computer and System Sciences*, vol. 73, no. 4, pp. 669–689.

Fuchs, L., U. Pankoke-Babatz, and W. Prinz (1995): ‘Supporting Cooperative Awareness with Local Event Mechanisms: The GroupDesk System’. In: H. Marmolin, Y. Sundblad, and K. Schmidt (eds.): *Proceedings of the Fourth European Conference on Computer-Supported Cooperative Work ECSCW ’95: 10–14 September, 1995, Stockholm, Sweden*. Dordrecht: Springer Netherlands, pp. 247–262.

Jeners, N., O. Lobunets, and W. Prinz (2013): ‘What groupware functionality do users really use? A study of collaboration within digital ecosystems’. In: *2013 7th IEEE International Conference on Digital Ecosystems and Technologies (DEST)*. pp. 49–54.

Jeners, N. and W. Prinz (2014): ‘Metrics for Cooperative Systems’. In: *Proceedings of the 18th International Conference on Supporting Group Work*. Sanibel Island, Florida, USA: ACM, pp. 91–99.

Jim, T., B. C. Pierce, and J. Vouillon (2002): ‘How to build a file synchronizer’. https://www.cis.upenn.edu/ bcpierce/courses/dd/papers/unisonimpl.ps.

Jokela, T., J. Ojala, and T. Olsson (2015): ‘A Diary Study on Combining Multiple Information Devices in Everyday Activities and Tasks’. In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. Seoul, Republic of Korea: ACM, pp. 3903–3912.

Kollmar, F. (2016): ‘The Cloud Storage Report – Dropbox Owns Cloud Storage on Mobile’. https://blog.cloudrail.com/cloud-storage-report-dropbox-owns-cloud-storage-mobile/.

Li, Q., L. Zhu, S. Zeng, and W. Q. Shang (eds.) (2012): ‘An Improved File System Synchronous Algorithm: 2012 Eighth International Conference on Computational Intelligence and Security’.

Lindholm, T., J. Kangasharju, and S. Tarkoma (2005): ‘A hybrid approach to optimistic file system directory tree synchronization’. In: V. Kumar, A. Zaslavsky, U. Cetintemel, and A. Labrinidis (eds.): *The 4th ACM international workshop on Data engineering for wireless and mobile access*. New York, NY, USA, pp. 49–56, ACM.

Microsoft Inc. (2018): ‘MSDN documentation: Naming Files, Paths, and Namespaces’. https://msdn.microsoft.com/en-us/library/windows/desktop/aa365247(v=vs.85).aspx.

Molli, P., G. Oster, H. Skaf-Molli, and A. Imine (2003): ‘Using the transformational approach to build a safe and generic data synchronizer’. In: *Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work*. Sanibel Island, Florida, USA: ACM, pp. 212–220.

Ng, A. and C. Sun (2016): 'Operational Transformation for Real-time Synchronization of Shared Workspace in Cloud Storage'. In: *Proceedings of the 19th International Conference on Supporting Group Work*. Sanibel Island, Florida, USA: ACM, pp. 61–70.

OrbiTeam Software GmbH & Co KG (2018): 'BSCW Social'. https://www.bscw.de/social/.

Phillips, W. G. (1999): 'Architectures for synchronous groupware: Technical Report 1999-425'.

Price, R. (2017): 'Google Drive now hosts more than 2 trillion files'. http://www.businessinsider.de/2-trillion-files-google-drive-exec-prabhakar-raghavan-2017-5.

Prinz, W., N. Jeners, R. Ruland, and M. Villa (2009): 'Supporting the Change of Cooperation Patterns by Integrated Collaboration Tools'. In: L. M. Camarinha-Matos, I. Paraskakis, and H. Afsarmanesh (eds.): *Leveraging Knowledge for Innovation in Collaborative Networks*. Berlin, Heidelberg, pp. 651–658, Springer Berlin Heidelberg.

Ramsey, N. and E. Csirmaz (2001): 'An algebraic approach to file synchronization'. In: A. M. Tjoa and V. Gruhn (eds.): *the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium*. p. 175.

Ridge, T., D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell (2015): 'SibylFS: Formal specification and oracle-based testing for POSIX and real-world file systems'. In: *Proceedings of the 25th Symposium on Operating Systems Principles*. Monterey, California: ACM, pp. 38–53.

Sun, C. and C. Ellis (1998): 'Operational transformation in real-time group editors: Issues, algorithms, and achievements'. In: *Proceedings of the 1998 ACM conference on Computer supported cooperative work*. Seattle, Washington, USA: ACM, pp. 59–68.

Sun, D. and C. Sun (2009): 'Context-Based Operational Transformation in Distributed Collaborative Editing Systems'. *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 10, pp. 1454–1470.

Tao, V., M. Shapiro, and V. Rancurel (2015): 'Merging Semantics for Conflict Updates in Geo-distributed File Systems'. In: *Proceedings of the 8th ACM International Systems and Storage Conference*. New York, NY, USA, pp. 10:1–10:12, ACM.

Uppoor, S., M. D. Flouris, and A. Bilas (2010): 'Cloud-based synchronization of distributed file system hierarchies'. In: *2010 IEEE International Conference On Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS)*. pp. 1–4.

Voida, S., W. K. Edwards, M. W. Newman, R. E. Grinter, and N. Ducheneaut (2006): 'Share and Share Alike: Exploring the User Interface Affordances of File Sharing'. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA, pp. 221–230, ACM.

Vonrueden, M. and W. Prinz (2007): 'Distributed Document Contexts in Cooperation Systems'. In: B. Kokinov, D. C. Richardson, T. R. Roth-Berghofer, and L. Vieu (eds.): *Modeling and Using Context*. Berlin, Heidelberg, pp. 507–516, Springer Berlin Heidelberg.

Wikipedia (2017a): 'Comparison of file synchronization software'.

Wikipedia (2017b): 'Comparison of file systems'.

Yang, C.-T., W.-C. Shih, C.-L. Huang, F.-C. Jiang, and W. C.-C. Chu (2016): 'On construction of a distributed data storage system in cloud'. *Computing*, vol. 98, no. 1, pp. 93–118.