Unison File Synchronizer

User Manual and Reference Guide

Version 2.54.0

Copyright 1998-2023, Benjamin C. Pierce

Contents

1 Overview **4**

2 Preface **5**
2.1 People . . . 5
2.2 Obtaining Unison . . . 5
2.3 Community, Maintenance, and Development . . . 5
2.4 Copying . . . 5
2.5 Acknowledgements . . . 5

3 Upgrading **6**
3.1 Version interoperability . . . 6

4 Tutorial **8**
4.1 Preliminaries . . . 8
4.2 Local Usage . . . 8
4.3 Remote Usage . . . 10
4.4 Remote Shell Method . . . 10
4.5 Socket Method . . . 11
4.5.1 TCP Sockets . . . 11
4.5.2 Unix Domain Sockets . . . 12
4.6 Using Unison for All Your Files . . . 12
4.7 Using Unison to Synchronize More Than Two Machines . . . 13
4.8 Going Further . . . 13

5 Basic Concepts **14**
5.1 Roots . . . 14
5.2 Paths . . . 15
5.3 What is an Update? . . . 15
5.4 What is a Conflict? . . . 16
5.5 Reconciliation . . . 16
5.6 Invariants . . . 16
5.7 Caveats and Shortcomings . . . 17

6 Reference Guide **19**
6.1 Running Unison . . . 19
6.2 The `.unison` Directory . . . 19
6.3 Archive Files . . . 19
6.4 Preferences . . . 20
6.5 Profiles . . . 33
6.6 Sample Profiles . . . 34
6.6.1 A Minimal Profile . . . 34
6.6.2 A Basic Profile . . . 34
6.6.3 A Power-User Profile . . . 35
6.7 Keeping Backups . . . 36
6.8 Merging Conflicting Versions . . . 37
6.9 The User Interface . . . 39
6.10 Interrupting a Synchronization . . . 39
6.11 Exit Code . . . 40
6.12 Path Specification . . . 40
6.13 Ignoring Paths . . . 41
6.14 Moved or Renamed Paths . . . 42
6.15 Symbolic Links . . . 43

6.16 Permissions 44
6.17 Access Control Lists - ACLs 44
6.18 Extended Attributes - xattrs 45
6.19 Cross-Platform Synchronization 46
6.20 Slow Links 46
6.21 Fast Update Detection 46
6.22 Mount Points and Removable Media 47

1 Overview

Unison is a file-synchronization tool for POSIX-compliant systems (e.g. BSDs, GNU/Linux, macOS) and Windows. It allows two replicas of a collection of files and directories to be stored on different hosts (or different disks on the same host), modified separately, and then brought up to date by propagating the changes in each replica to the other.

Features:

- Unison works *across* platforms, allowing you to synchronize a Windows laptop with a Unix server, for example.
- Unlike a distributed filesystem, Unison is a user-level program: there is no need to modify the kernel or to have superuser privileges on either host.
- Unlike simple mirroring or backup utilities, Unison can deal with updates to both replicas of a distributed directory structure. Updates that do not conflict can be propagated automatically. Conflicting updates are detected and displayed.
- Unison works between any pair of machines connected to the internet, communicating over either a direct socket link or tunneling over an encrypted `ssh` connection. It is careful with network bandwidth, and runs well over slow links. Transfers of small updates to large files are optimized using a compression protocol similar to rsync.
- Unison has a clear and precise specification, described below.
- Unison is resilient to failure. It is careful to leave the replicas and its own private structures in a sensible state at all times, even in case of abnormal termination or communication failures.
- Unison is free; full source code is available under the GNU Public License.

2 Preface

2.1 People

Benjamin Pierce leads the Unison project. The current version of Unison was designed and implemented by Trevor Jim, Benjamin Pierce, and Jérôme Vouillon, with Alan Schmitt, Malo Denielou, Zhe Yang, Sylvain Gommier, and Matthieu Goulay. The Mac user interface was started by Trevor Jim and enormously improved by Ben Willmore. Our implementation of the rsync protocol was built by Norman Ramsey and Sylvain Gommier. It is based on Andrew Tridgell's thesis work and inspired by his rsync utility. The mirroring and merging functionality was implemented by Sylvain Roy, improved by Malo Denielou, and improved yet further by Stéphane Lescuyer. Jacques Garrigue contributed the original Gtk version of the user interface; the Gtk2 version was built by Stephen Tse. Sundar Balasubramaniam helped build a prototype implementation of an earlier synchronizer in Java. Insik Shin and Insup Lee contributed design ideas to this implementation. Cedric Fournet contributed to an even earlier prototype.

2.2 Obtaining Unison

Source code Unison is primarily distributed as source code, which contains instructions in `INSTALL.md`:

`https://github.com/bcpierce00/unison`

Binaries The Unison wiki contains information about builds done as part of Continuous Integration and other sources of binaries; read the entire wiki at:

`https://github.com/bcpierce00/unison/wiki`

2.3 Community, Maintenance, and Development

Many people use and contribute to Unison. This community has two main homes.

Mailinglists Most discussion is appropriate on one of the mailinglists:

`https://github.com/bcpierce00/unison/wiki/Mailing-Lists`

Reporting Bugs Bug reports and feature requests may be made after reading the guidelines:

`https://github.com/bcpierce00/unison/wiki/Reporting-Bugs-and-Feature-Requests`

Help improving Unison is welcome; see `CONTRIBUTING.md` in the sources.

2.4 Copying

This file is part of Unison.

Unison is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

Unison is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

The GNU General Public License can be found at `http://www.gnu.org/licenses`. A copy is also included in the Unison source distribution in the file `COPYING`.

2.5 Acknowledgements

Work on Unison has been supported by the National Science Foundation under grants CCR-9701826 and ITR-0113226, *Principles and Practice of Synchronization*, and by University of Pennsylvania's Institute for Research in Cognitive Science (IRCS).

3 Upgrading

(This section is perhaps misplaced, but is early because it is far better to have at least skimmed it than to not know it exists.)

Before upgrading, it is a good idea to run the *old* version one last time, to make sure all your replicas are completely synchronized. A new version of Unison will sometimes introduce a different format for the archive files used to remember information about the previous state of the replicas. In this case, the old archive will be ignored (not deleted — if you roll back to the previous version of Unison, you will find the old archives intact), which means that any differences between the replicas will show up as conflicts that need to be resolved manually.

As of version 2.52, Unison has a degree of backward and forward compatibility. This means three things. First, it is possible for local and remote machines to run a different version of Unison. Second, it is possible for local and remote machines to run a version (same or different) of Unison built with a different version of OCaml compiler (this has been problematic historically). Lastly, it is possible to upgrade Unison on the local machine (compiled with any OCaml version) and keep the existing archive.

Upgrading from Unison 2.48 or 2.51: If version interoperability requirements are followed then Unison 2.52 up to 2.53.8 can upgrade the archive created by Unison 2.48 to 2.51. To avoid rebuilding archive files when upgrading from a version older than 2.52, you must install version 2.52 or newer built with the same OCaml version as your previous version of Unison, and then run it at least once on each root. Doing so will upgrade the archive file. Upgrading directly to version newer than 2.53.8 is not supported; upgrade first to a version between 2.52 and 2.53.8 if you want to keep the archives.

After upgrading the archive, you are free to swap the Unison 2.52 or newer executable to one compiled with a different version of OCaml. The archive file is no longer dependent on the compiler version.

3.1 Version interoperability

To ensure interoperability with different Unison versions on local and remote machines, and to upgrade from an earlier version *without rebuilding the archive files*, you have to remember these guidelines. Upgrading from an incompatible version, while possible and normal, will require fully scanning both roots, which can be time-consuming with big replicas.

Unison 2.52 and newer are compatible with:

- *Unison 2.52 or newer* (for as long as backwards compatibility is maintained in the newer versions). You do not have to pay any attention to OCaml compiler versions.

Additionally, **Unison 2.52 up to 2.53.8 (included)** are compatible with:

- *Unison 2.51* if both versions are compiled with same OCaml compiler version (you can see which compiler version was used by running `unison -version`).
- *Unison 2.48* if both versions are compiled with same OCaml compiler version. See special notes below.

Interoperability matrix for quick reference:

Client versions	Server versions		
	newer than 2.53.8	2.52 to 2.53.8	older than 2.52
newer than 2.53.8	full interop	full interop	**no interop**
2.52 to 2.53.8	full interop	full interop	*see below*
older than 2.52	**no interop**	*see below*	*see below*

Client versions	Server versions		
	2.52 to 2.53.8	2.51	2.48
2.52 to 2.53.8	full interop	same OCaml version	same OCaml version
2.51	same OCaml version	full interop	no interop
2.48	same OCaml version*	no interop	full interop

Special notes for Unison 2.48:

- Unison 2.48 does not show which OCaml compiler was used to compile it. If you do not have the option of re-compiling the 2.48 version, you have two alternatives. First (and most likely to succeed), see what is the version of the OCaml compiler in the same package repository where you installed Unison 2.48 from, then use Unison 2.52 compiled with that version. Second, you can just try Unison 2.52 executables compiled with different OCaml versions and see which one works with your copy of Unison 2.48.
- When running Unison 2.48 on the client machine with Unison 2.52 or newer on the server machine, you have to do some additional configuration. The Unison executable name on the server must start with `unison-2.48` (just `unison-2.48` is ok, as is `unison-2.48.exe`, but also `unison-2.48+ocaml-4.05`). If using TCP socket connection to the server then you're all set! If using `ssh` then you have to add one of the following options to your profile or as a command-line argument on the client machine: `-addversionno`; see Section 4.3 [Remote Usage], or `-servercmd`; see Section 4.4 [Remote Shell Method].

4 Tutorial

4.1 Preliminaries

Unison can be used with either of two user interfaces:

1. a textual interface and
2. a graphical interface

The textual interface is more convenient for running from scripts and works on dumb terminals; the graphical interface is better for most interactive use. For this tutorial, you can use either. If you are running Unison from the command line, just typing `unison` will select either the text or the graphical interface, depending on which has been selected as default when the executable you are running was built. You can force the text interface even if graphical is the default by adding `-ui text`. The other command-line arguments to both versions are identical.

The graphical version can also be run directly by clicking on its icon. For this tutorial, we assume that you're starting it from the command line.

Unison can synchronize files and directories on a single machine, or between two machines on a network. (The same program runs on both machines; the only difference is which one is responsible for displaying the user interface.) If you're only interested in a single-machine setup, then let's call that machine the *client*. If you're synchronizing two machines, let's call them *client* and *server*.

4.2 Local Usage

Let's get the client machine set up first and see how to synchronize two directories on a single machine.

Ensure that unison is installed on your system.

Create a small test directory `a.tmp` containing a couple of files and/or subdirectories, e.g.,

```
mkdir a.tmp
touch a.tmp/a a.tmp/b
mkdir a.tmp/d
touch a.tmp/d/f
```

Copy this directory to b.tmp:

```
cp -r a.tmp b.tmp
```

Now try synchronizing `a.tmp` and `b.tmp`. (Since they are identical, synchronizing them won't propagate any changes, but Unison will remember the current state of both directories so that it will be able to tell next time what has changed.) Type:

```
unison a.tmp b.tmp
```

(You may need to add `-ui text`, depending how your unison binary was built.)

Textual Interface:

You should see a message notifying you that all the files are actually equal and then get returned to the command line.

Graphical Interface:

You should get a big empty window with a message at the bottom notifying you that all files are identical. Choose the Exit item from the File menu to get back to the command line.

Next, make some changes in a.tmp and/or b.tmp. For example:

```
rm a.tmp/a
echo "Hello" > a.tmp/b
echo "Hello" > b.tmp/b
date > b.tmp/c
echo "Hi there" > a.tmp/d/h
echo "Hello there" > b.tmp/d/h
```

Run Unison again:

```
unison a.tmp b.tmp
```

This time, the user interface will display only the files that have changed. If a file has been modified in just one replica, then it will be displayed with an arrow indicating the direction that the change needs to be propagated. For example,

```
                <---  new file   c  [f]
```

indicates that the file `c` has been modified only in the second replica, and that the default action is therefore to propagate the new version to the first replica. To **f**ollow Unison's recommendation, press the "f" at the prompt.

If both replicas are modified and their contents are different, then the changes are in conflict: `<-?->` is displayed to indicate that Unison needs guidance on which replica should override the other.

```
  new file  <-?->  new file   d/h  []
```

By default, neither version will be propagated and both replicas will remain as they are.

If both replicas have been modified but their new contents are the same (as with the file `b`), then no propagation is necessary and nothing is shown. Unison simply notes that the file is up to date.

These display conventions are used by both versions of the user interface. The only difference lies in the way in which Unison's default actions are either accepted or overridden by the user.

Textual Interface:

The status of each modified file is displayed, in turn. When the copies of a file in the two replicas are not identical, the user interface will ask for instructions as to how to propagate the change. If some default action is indicated (by an arrow), you can simply press Return to go on to the next changed file. If you want to do something different with this file, press "`<`" or "`>`" to force the change to be propagated from right to left or from left to right, or else press "`/`" to skip this file and leave both replicas alone. When it reaches the end of the list of modified files, Unison will ask you one more time whether it should proceed with the updates that have been selected.

When Unison stops to wait for input from the user, pressing "`?`" will always give a list of possible responses and their meanings.

Graphical Interface:

The main window shows all the files that have been modified in either `a.tmp` or `b.tmp`. To override a default action (or to select an action in the case when there is no default), first select the file, either by clicking on its name or by using the up- and down-arrow keys. Then press either the left-arrow or "`<`" key (to cause the version in b.tmp to propagate to a.tmp) or the right-arrow or "`>`" key (which makes the a.tmp version override b.tmp).

Every keyboard command can also be invoked from the menus at the top of the user interface. (Conversely, each menu item is annotated with its keyboard equivalent, if it has one.)

When you are satisfied with the directions for the propagation of changes as shown in the main window, click the "Go" button to set them in motion. A check sign will be displayed next to each filename when the file has been dealt with.

4.3 Remote Usage

Next, we'll get Unison set up to synchronize replicas on two different machines.

NB: Unison has not been designed to run with elevated privileges (e.g. setuid), and it has not been audited for that environment. Therefore Unison should be run with the userid of the owner of the files to be synchronized, and should never be run setuid or similar. (Problems encountered when running setuid etc. must be reproduced without setuid before being reported as bugs.)

Follow the instructions in the Installation section to download or build an executable version of Unison on the server machine, and install it somewhere on your search path. (It doesn't matter whether you install the textual or graphical version, since the copy of Unison on the server doesn't need to display any user interface at all. The major benefit of installing the textual version is that it doesn't have any external dependencies required by the GUI executable.)

It is important that the version of Unison installed on the server machine is the same as the version of Unison on the client machine. But some flexibility on the version of Unison at the client side can be achieved by using the `-addversionno` option; see Section 6.4 [Preferences].

Now there is a decision to be made. Unison provides two methods for communicating between the client and the server:

- *Remote shell method*: To use this method, you must have some way of invoking remote commands on the server from the client's command line, using a facility such as `ssh`. This method is more convenient (since there is no need to manually start a "unison server" process on the server) and also more secure, assuming you are using `ssh`).
- *TCP socket method*: This method requires only that you can get TCP packets from the client to the server and back. It is insecure and should not be used.
- *Unix socket method*: This method only works within a single machine. It is similar to the TCP sockets method, but it is possible to configure it securely.

Decide which of these you want to try, and continue with Section 4.4 [Remote Shell Method] or Section 4.5 [Socket Method], as appropriate.

4.4 Remote Shell Method

The standard remote shell facility on Unix systems is `ssh`.

Running `ssh` requires some coordination between the client and server machines to establish that the client is allowed to invoke commands on the server; please refer to the `ssh` documentation for information on how to set this up.

First, test that we can invoke Unison on the server from the client. Typing

```
ssh remotehostname unison -version
```

should print the same version information as running

```
unison -version
```

locally on the client. If remote execution fails, then either something is wrong with your ssh setup (e.g., "permission denied") or else the search path that's being used when executing commands on the server doesn't contain the `unison` executable (e.g., "command not found").

Create a test directory `a.tmp` in your home directory on the client machine.

Test that the local unison client can start and connect to the remote server. Type

```
unison -testServer a.tmp ssh://remotehostname/a.tmp
```

Now cd to your home directory and type:

```
unison a.tmp ssh://remotehostname/a.tmp
```

The result should be that the entire directory `a.tmp` is propagated from the client to your home directory on the server.

After finishing the first synchronization, change a few files and try synchronizing again. You should see similar results as in the local case.

If your user name on the server is not the same as on the client, you need to specify it on the command line:

```
unison a.tmp ssh://username@remotehostname/a.tmp
```

Notes:

- If you want to put `a.tmp` some place other than your home directory on the remote host, you can give an absolute path for it by adding an extra slash between `remotehostname` and the beginning of the path:

  ```
  unison a.tmp ssh://remotehostname//absolute/path/to/a.tmp
  ```

- You can give an explicit path for the `unison` executable on the server by using the command-line option `-servercmd /full/path/name/of/unison` or adding `servercmd=/full/path/name/of/unison` to your profile (see Section 6.5 [Profiles]). Similarly, you can specify an explicit path for the `ssh` program using the `-sshcmd` option. Extra arguments can be passed to `ssh` by setting the `-sshargs` preference.

- By leveraging `-sshcmd` and `-sshargs`, you can effectively use any remote shell program, not just `ssh`; just remember that the roots are still specified with `ssh` as the protocol, that is, they have to start with `ssh://`.

4.5 Socket Method

To run Unison over a socket connection, you must start a Unison daemon process on the server. This process runs continuously, waiting for connections over a given socket from client machines running Unison and processing their requests in turn.

Since the socket method is not used by many people, its functionality is rather limited. For example, the server can only deal with one client at a time.

Note that the Unison daemon process is always started with a command-line argument; not from a profile.

4.5.1 TCP Sockets

> **Warning:** The TCP socket method is insecure: not only are the texts of your changes transmitted over the network in unprotected form, it is also possible for anyone in the world to connect to the server process and read out the contents of your filesystem! (Of course, to do this they must understand the protocol that Unison uses to communicate between client and server, but all they need for this is a copy of the Unison sources.) The socket method is provided only for expert users with specific needs; everyone else should use the `ssh` method.

To start the daemon for connections over a TCP socket, type

```
unison -socket NNNN
```

on the server machine, where `NNNN` is the TCP port number that the daemon should listen on for connections from clients. (`NNNN` can be any large number that is not being used by some other program; if `NNNN` is already in use, Unison will exit with an error message.)

Create a test directory `a.tmp` in your home directory on the client machine. Now type:

```
unison a.tmp socket://remotehostname:NNNN/a.tmp
```

Note that paths specified by the client will be interpreted relative to the directory in which you start the server process; this behavior is different from the ssh case, where the path is relative to your home directory on the server. The result should be that the entire directory `a.tmp` is propagated from the client to the server (`a.tmp` will be created on the server in the directory that the server was started from). After finishing the first synchronization, change a few files and try synchronizing again. You should see similar results as in the local case.

By default Unison will listen for incoming connections on all interfaces. If you want to limit this to certain interfaces or addresses then you can use the `-listen` command-line argument, specifying a host name or an IP address to listen on. `-listen` can be given multiple times to listen on several addresses.

4.5.2 Unix Domain Sockets

To start the daemon for connections over a Unix domain socket, type

```
unison -socket PPPP
```

where `PPPP` is the path to a Unix socket that the daemon should open for connections from clients. (`PPPP` can be any absolute or relative path the server process has access to but it must not exist yet; the socket is created at that path when the daemon process is started.) You are responsible for securing access to the socket path. For example, this can be done by controlling the permissions of socket's parent directory, or ensuring a restrictive `umask` value when starting Unison.

Clients can connect to a server over a Unix domain socket by specifying the absolute or relative path to the socket, instead of a server address and port number:

```
unison a.tmp socket://{path/to/unix/socket}/a.tmp
```

(socket path is enclosed in curly braces).

Note that Unix domain sockets are local sockets (they exist in the filesystem namespace). One could use Unixs socket remotely, by forwarding access to the socket by other means, for example by using `spiped` secure pipe daemon.

4.6 Using Unison for All Your Files

Once you are comfortable with the basic operation of Unison, you may find yourself wanting to use it regularly to synchronize your commonly used files. There are several possible ways of going about this:

1. Synchronize your whole home directory, using the Ignore facility (see Section 6.13 [Ignoring Paths]) to avoid synchronizing temporary files and things that only belong on one host.

2. Create a subdirectory called `shared` (or `current`, or whatever) in your home directory on each host, and put all the files you want to synchronize into this directory.

3. Make your home directory the root of the synchronization, but tell Unison to synchronize only some of the files and subdirectories within it on any given run. This can be accomplished by using the `-path` switch on the command line:

   ```
   unison /home/username ssh://remotehost//home/username -path shared
   ```

   The `-path` option can be used as many times as needed, to synchronize several files or subdirectories:

   ```
   unison /home/username ssh://remotehost//home/username \
      -path shared \
      -path pub \
      -path .netscape/bookmarks.html
   ```

   These `-path` arguments can also be put in your preference file. See Section 6.4 [Preferences] for an example.

Most people find that they only need to maintain a profile (or profiles) on one of the hosts that they synchronize, since Unison is always initiated from this host. (For example, if you're synchronizing a laptop with a fileserver, you'll probably always run Unison on the laptop.) This is a bit different from the usual situation with asymmetric mirroring programs like `rdist`, where the mirroring operation typically needs to be initiated from the machine with the most recent changes. Section 6.5 [Profiles] covers the syntax of Unison profiles, together with some sample profiles.

4.7 Using Unison to Synchronize More Than Two Machines

Unison is designed for synchronizing pairs of replicas. However, it is possible to use it to keep larger groups of machines in sync by performing multiple pairwise synchronizations.

If you need to do this, the most reliable way to set things up is to organize the machines into a "star topology," with one machine designated as the "hub" and the rest as "spokes," and with each spoke machine synchronizing only with the hub. The big advantage of the star topology is that it eliminates the possibility of confusing "spurious conflicts" arising from the fact that a separate archive is maintained by Unison for every pair of hosts that it synchronizes.

4.8 Going Further

On-line documentation for the various features of Unison can be obtained either by typing

```
unison -doc topics
```

at the command line, or by selecting the Help menu in the graphical user interface. The on-line information and the printed manual are essentially identical.

If you use Unison regularly, you should subscribe to one of the mailing lists, to receive announcements of new versions. See Section 2.2 [Obtaining Unison].

5 Basic Concepts

To understand how Unison works, it is necessary to discuss a few straightforward concepts.

These concepts are developed more rigorously and at more length in a number of papers, available at `http://www.cis.upenn.edu/~bcpierce/papers`. But the informal presentation here should be enough for most users.

5.1 Roots

A replica's *root* tells Unison where to find a set of files to be synchronized, either on the local machine or on a remote host. For example,

relative/path/of/root

specifies a local root relative to the directory where Unison is started, while

/absolute/path/of/root

specifies a root relative to the top of the local filesystem, independent of where Unison is running. Remote roots can begin with `ssh://` to indicate that the remote server should be started with ssh:

`ssh://`*remotehost*`//`*absolute/path/of/root*
`ssh://`*user*`@`*remotehost*`/`*relative/path/of/root*

If the remote server is already running (in the socket mode), then the syntax

`socket://`*remotehost*`:`*portnum*`//`*absolute/path/of/root*
`socket://`*remotehost*`:`*portnum*`/`*relative/path/of/root*
`socket://[`*IPv6literal*`]:`*portnum*`/`*path*

is used to specify the hostname and the port that the client Unison should use to contact it. Syntax

`socket://{`*path/of/socket*`}//`*absolute/path/of/root*
`socket://{`*path/of/socket*`}/`*relative/path/of/root*

is used to specify the Unix domain socket the client Unison should use to contact the server.

The syntax for roots is based on that of URIs (described in RFC 2396). The full grammar is:

```
replica ::= [protocol:]//[user@][host][:port][/path]
          |  path

protocol ::= file
           |  socket
           |  ssh

user ::= [-_a-zA-Z0-9%@]+

host ::= [-_a-zA-Z0-9.]+
       |  \[ [a-f0-9:.]+ zone? \]    IPv6 literals (no future format).
       |  { [^}]+ }                   For Unix domain sockets only.

zone ::= %[-_a-zA-Z0-9~%.]+

port ::= [0-9]+
```

When `path` is given without any protocol prefix, the protocol is assumed to be `file:`. Under Windows, it is possible to synchronize with a remote directory using the `file:` protocol over the Windows Network Neighborhood. For example,

```
unison foo //host/drive/bar
```

synchronizes the local directory `foo` with the directory `drive:\bar` on the machine `host`, provided that `host` is accessible via Network Neighborhood. When the `file:` protocol is used in this way, there is no need for a Unison server to be running on the remote host. However, running Unison this way is only a good idea if the remote host is reached by a very fast network connection, since the full contents of every file in the remote replica will have to be transferred to the local machine to detect updates.

The names of roots are *canonized* by Unison before it uses them to compute the names of the corresponding archive files, so `//saul//home/bcpierce/common` and `//saul.cis.upenn.edu/common` will be recognized as the same replica under different names.

5.2 Paths

A *path* refers to a point *within* a set of files being synchronized; it is specified relative to the root of the replica.

Formally, a path is just a sequence of names, separated by `/`. Note that the path separator character is always a forward slash, no matter what operating system Unison is running on. Forward slashes are converted to backslashes as necessary when paths are converted to filenames in the local filesystem on a particular host. (For example, suppose that we run Unison on a Windows system, synchronizing the local root `c:\pierce` with the root `ssh://saul.cis.upenn.edu/home/bcpierce` on a Unix server. Then the path `current/todo.txt` refers to the file `c:\pierce\current\todo.txt` on the client and `/home/bcpierce/current/todo.txt` on the server.)

The empty path (i.e., the empty sequence of names) denotes the whole replica. Unison displays the empty path as "`[root]`."

If `p` is a path and `q` is a path beginning with `p`, then `q` is said to be a *descendant* of `p`. (Each path is also a descendant of itself.)

5.3 What is an Update?

The *contents* of a path `p` in a particular replica could be a file, a directory, a symbolic link, or absent (if `p` does not refer to anything at all in that replica). More specifically:

- If `p` refers to an ordinary file, then the contents of `p` are the actual contents of this file (a string of bytes) plus the current permission bits of the file.
- If `p` refers to a symbolic link, then the contents of `p` are just the string specifying where the link points.
- If `p` refers to a directory, then the contents of `p` are just the token "DIRECTORY" plus the current permission bits of the directory.
- If `p` does not refer to anything in this replica, then the contents of `p` are the token "ABSENT."

Unison keeps a record of the contents of each path after each successful synchronization of that path (i.e., it remembers the contents at the last moment when they were the same in the two replicas).

We say that a path is *updated* (in some replica) if its current contents are different from its contents the last time it was successfully synchronized. Note that whether a path is updated has nothing to do with its last modification time—Unison considers only the contents when determining whether an update has occurred. This means that touching a file without changing its contents will *not* be recognized as an update. A file can even be changed several times and then changed back to its original contents; as long as Unison is only run at the end of this process, no update will be recognized.

What Unison actually calculates is a close approximation to this definition; see Section 5.7 [Caveats and Shortcomings].

5.4 What is a Conflict?

A path is said to be *conflicting* if the following conditions all hold:

1. it has been updated in one replica,
2. it or any of its descendants has been updated in the other replica, and
3. its contents in the two replicas are not identical.

5.5 Reconciliation

Unison operates in several distinct stages:

1. On each host, it compares its archive file (which records the state of each path in the replica when it was last synchronized) with the current contents of the replica, to determine which paths have been updated.
2. It checks for "false conflicts" — paths that have been updated on both replicas, but whose current values are identical. These paths are silently marked as synchronized in the archive files in both replicas.
3. It displays all the updated paths to the user. For updates that do not conflict, it suggests a default action (propagating the new contents from the updated replica to the other). Conflicting updates are just displayed. The user is given an opportunity to examine the current state of affairs, change the default actions for nonconflicting updates, and choose actions for conflicting updates.
4. It performs the selected actions, one at a time. Each action is performed by first transferring the new contents to a temporary file on the receiving host, then atomically moving them into place.
5. It updates its archive files to reflect the new state of the replicas.

5.6 Invariants

Given the importance and delicacy of the job that it performs, it is important to understand both what a synchronizer does under normal conditions and what can happen under unusual conditions such as system crashes and communication failures.

Unison is careful to protect both its internal state and the state of the replicas at every point in this process. Specifically, the following guarantees are enforced:

- At every moment, each path in each replica has either (1) its *original* contents (i.e., no change at all has been made to this path), or (2) its *correct* final contents (i.e., the value that the user expected to be propagated from the other replica).
- At every moment, the information stored on disk about Unison's private state can be either (1) unchanged, or (2) updated to reflect those paths that have been successfully synchronized.

The upshot is that it is safe to interrupt Unison at any time, either manually or accidentally. [Caveat: the above is *almost* true there are occasionally brief periods where it is not (and, because of shortcoming of the Posix filesystem API, cannot be); in particular, when it is copying a file onto a directory or vice versa, it must first move the original contents out of the way. If Unison gets interrupted during one of these periods, some manual cleanup may be required. In this case, a file called `DANGER.README` will be left in the `.unison` directory, containing information about the operation that was interrupted. The next time you try to run Unison, it will notice this file and warn you about it.]

If an interruption happens while it is propagating updates, then there may be some paths for which an update has been propagated but which have not been marked as synchronized in Unison's archives. This is no problem: the next time Unison runs, it will detect changes to these paths in both replicas, notice that

the contents are now equal, and mark the paths as successfully updated when it writes back its private state at the end of this run.

If Unison is interrupted, it may sometimes leave temporary working files (with suffix `.tmp`) in the replicas. It is safe to delete these files. Also, if the `backups` flag is set, Unison will leave around old versions of files that it overwrites, with names like `file.0.unison.bak`. These can be deleted safely when they are no longer wanted.

Unison is not bothered by clock skew between the different hosts on which it is running. It only performs comparisons between timestamps obtained from the same host, and the only assumption it makes about them is that the clock on each system always runs forward.

If Unison finds that its archive files have been deleted (or that the archive format has changed and they cannot be read, or that they don't exist because this is the first run of Unison on these particular roots), it takes a conservative approach: it behaves as though the replicas had both been completely empty at the point of the last synchronization. The effect of this is that, on the first run, files that exist in only one replica will be propagated to the other, while files that exist in both replicas but are unequal will be marked as conflicting.

Touching a file without changing its contents should never affect whether or not Unison does an update. (When running with the fastcheck preference set to true—the default on Unix systems—Unison uses file modtimes for a quick first pass to tell which files have definitely not changed; then, for each file that might have changed, it computes a fingerprint of the file's contents and compares it against the last-synchronized contents. Also, the `-times` option allows you to synchronize file times, but it does not cause identical files to be changed; Unison will only modify the file times.)

It is safe to "brainwash" Unison by deleting its archive files *on both replicas*. The next time it runs, it will assume that all the files it sees in the replicas are new.

It is safe to modify files while Unison is working. If Unison discovers that it has propagated an out-of-date change, or that the file it is updating has changed on the target replica, it will signal a failure for that file. Run Unison again to propagate the latest change.

Changes to the ignore patterns from the user interface (e.g., using the 'i' key) are immediately reflected in the current profile.

5.7 Caveats and Shortcomings

Here are some things to be careful of when using Unison.

- In the interests of speed, the update detection algorithm may (depending on which OS architecture that you run Unison on) actually use an approximation to the definition given in Section 5.3 [What is an Update?].

  In particular, the Unix implementation does not compare the actual contents of files to their previous contents, but simply looks at each file's inode number and modtime; if neither of these have changed, then it concludes that the file has not been changed.

  Under normal circumstances, this approximation is safe, in the sense that it may sometimes detect "false updates" but will never miss a real one. However, it is possible to fool it, for example by using `retouch` to change a file's modtime back to a time in the past.

- If you synchronize between a single-user filesystem and a shared Unix server, you should pay attention to your permission bits: by default, Unison will synchronize permissions verbatim, which may leave group-writable files on the server that could be written over by a lot of people.

  You can control this by setting your `umask` on both computers to something like 022, masking out the "world write" and "group write" permission bits.

  Unison does not synchronize the `setuid` and `setgid` bits, for security.

- The graphical user interface is single-threaded. This means that if Unison is performing some long-running operation, the display will not be repainted until it finishes. We recommend not trying to do anything with the user interface while Unison is in the middle of detecting changes or propagating files.

- Unison does not understand hard links.
- It is important to be a little careful when renaming directories containing `ignore`d files.

  For example, suppose Unison is synchronizing directory A between the two machines called the "local" and the "remote" machine; suppose directory A contains a subdirectory D; and suppose D on the local machine contains a file or subdirectory P that matches an ignore directive in the profile used to synchronize. Thus path A/D/P exists on the local machine but not on the remote machine.

  If D is renamed to D' on the remote machine, and this change is propagated to the local machine, all such files or subdirectories P will be deleted. This is because Unison sees the rename as a delete and a separate create: it deletes the old directory (including the ignored files) and creates a new one (*not* including the ignored files, since they are completely invisible to it).

6 Reference Guide

This section covers the features of Unison in detail.

6.1 Running Unison

There are several ways to start Unison.

- Typing "`unison` *`profile`*" on the command line. Unison will look for a file *`profile`*`.prf` in the `.unison` directory. If this file does not specify a pair of roots, Unison will prompt for them and add them to the information specified by the profile.
- Typing "`unison` *`profile root1 root2`*" on the command line. In this case, Unison will use *`profile`*, which should not contain any `root` directives.
- Typing "`unison` *`root1 root2`*" on the command line. This has the same effect as typing "`unison default` *`root1 root2`*."
- Typing just "`unison`" (or invoking Unison by clicking on a desktop icon). In this case, Unison will ask for the profile to use for synchronization (or create a new one, if necessary).

6.2 The `.unison` Directory

Unison stores a variety of information in a private directory on each host. If the environment variable `UNISON` is defined, then its value will be used as the path/folder name for this directory. This can be just a name, or a path.

A name on it's own, for example `UNISON=mytestname` will place a folder in the same directory that the Unison binary was run in, with that name. Using a path like `UNISON=../mytestname2` will place that folder in the folder above where the Unison binary was run from.

If `UNISON` is not defined, then the directory depends on which operating system you are using. In Unix, the default is to use `$HOME/.unison`. In Windows, if the environment variable `USERPROFILE` is defined, then the directory will be `$USERPROFILE\.unison`; otherwise if `HOME` is defined, it will be `$HOME\.unison`; otherwise, it will be `c:\.unison`. On OS X, `$HOME/.unison` will be used if it is present, but `$HOME/Library/Application Support/Unison` will be created and used by default.

The archive file for each replica is found in the `.unison` directory on that replica's host. Profiles (described below) are always taken from the `.unison` directory on the client host.

Note that Unison maintains a completely different set of archive files for each pair of roots.

We do not recommend synchronizing the whole `.unison` directory, as this will involve frequent propagation of large archive files. It should be safe to do it, though, if you really want to. Synchronizing just the profile files in the `.unison` directory is definitely OK.

6.3 Archive Files

The name of the archive file on each replica is calculated from

- the *canonical names* of all the hosts (short names like `saul` are converted into full addresses like `saul.cis.upenn.edu`),
- the paths to the replicas on all the hosts (again, relative pathnames, symbolic links, etc. are converted into full, absolute paths), and
- an internal version number that is changed whenever a new Unison release changes the format of the information stored in the archive.

This method should work well for most users. However, it is occasionally useful to change the way archive names are generated. Unison provides two ways of doing this.

The function that finds the canonical hostname of the local host (which is used, for example, in calculating the name of the archive file used to remember which files have been synchronized) normally uses the `gethostname` operating system call. However, if the environment variable `UNISONLOCALHOSTNAME` is set, its value will be used instead. This makes it easier to use Unison in situations where a machine's name changes frequently (e.g., because it is a laptop and gets moved around a lot).

A more powerful way of changing archive names is provided by the `rootalias` preference. The preference file may contain any number of lines of the form:

```
rootalias = //hostnameA//path-to-replicaA -> //hostnameB/path-to-replicaB
```

When calculating the name of the archive files for a given pair of roots, Unison replaces any root that matches the left-hand side of any rootalias rule by the corresponding right-hand side.

So, if you need to relocate a root on one of the hosts, you can add a rule of the form:

```
rootalias = //new-hostname//new-path -> //old-hostname/old-path
```

Note that root aliases are case-sensitive, even on case-insensitive file systems.

Warning: The `rootalias` option is dangerous and should only be used if you are sure you know what you're doing. In particular, it should only be used if you are positive that either (1) both the original root and the new alias refer to the same set of files, or (2) the files have been relocated so that the original name is now invalid and will never be used again. (If the original root and the alias refer to different sets of files, Unison's update detector could get confused.) After introducing a new `rootalias`, it is a good idea to run Unison a few times interactively (with the `batch` flag off, etc.) and carefully check that things look reasonable—in particular, that update detection is working as expected.

6.4 Preferences

Many details of Unison's behavior are configurable by user-settable "preferences."

Some preferences are boolean-valued; these are often called *flags*. Others take numeric or string arguments, indicated in the preferences list by `n` or `xxx`. Some string arguments take the backslash as an escape to include the next character literally; this is mostly useful to escape a space or the backslash; a trailing backslash is ignored and is useful to protect a trailing whitespace in the string that would otherwise be trimmed. Most of the string preferences can be given several times; the arguments are accumulated into a list internally.

There are two ways to set the values of preferences: temporarily, by providing command-line arguments to a particular run of Unison, or permanently, by adding commands to a *profile* in the `.unison` directory on the client host. The order of preferences (either on the command line or in preference files) is not significant. On the command line, preferences and other arguments (the profile name and roots) can be intermixed in any order.

To set the value of a preference `p` from the command line, add an argument `-p` (for a boolean flag) or `-p n` or `-p xxx` (for a numeric or string preference) anywhere on the command line. To set a boolean flag to `false` on the command line, use `-p=false`.

Here are all the preferences supported by Unison. This list can be obtained by typing `unison -help`.

```
Usage: unison [options]
    or unison root1 root2 [options]
    or unison profilename [options]

Basic options:

  General:
   -doc xxx            show documentation ('-doc topics' lists topics)
   -version            print version and exit

  What to sync:
```

```
  -group              synchronize group attributes
  -ignore xxx         add a pattern to the ignore list
  -ignorenot xxx      add a pattern to the ignorenot list
  -nocreation xxx     prevent file creations on one replica
  -nodeletion xxx     prevent file deletions on one replica
  -noupdate xxx       prevent file updates and deletions on one replica
  -owner              synchronize owner
  -path xxx           path to synchronize
  -perms n            part of the permissions which is synchronized
  -root xxx           root of a replica (should be used exactly twice)
  -times              synchronize modification times

 How to sync:
  -batch              batch mode: ask no questions at all

 How to sync (text interface (CLI) only):
  -auto               automatically accept default (nonconflicting) actions
  -silent             print nothing except error messages
  -terse              suppress status messages

 Text interface (CLI):
  -i                  interactive profile mode (text UI); command-line only

Advanced options:

 Fine-tune sync:
  -acl                synchronize ACLs
  -atomic xxx         add a pattern to the atomic list
  -follow xxx         add a pattern to the follow list
  -force xxx          force changes from this replica to the other
  -forcepartial xxx   add a pattern to the forcepartial list
  -ignorecase xxx     identify upper/lowercase filenames (true/false/default)
  -immutable xxx      add a pattern to the immutable list
  -immutablenot xxx   add a pattern to the immutablenot list
  -links xxx          allow the synchronization of symbolic links
                      (true/false/default)
  -merge xxx          add a pattern to the merge list
  -moves-experimental optimize transfers by detecting renames and moves
  -nocreationpartial xxx add a pattern to the nocreationpartial list
  -nodeletionpartial xxx add a pattern to the nodeletionpartial list
  -noupdatepartial xxx add a pattern to the noupdatepartial list
  -prefer xxx         choose this replica's version for conflicting changes
  -preferpartial xxx  add a pattern to the preferpartial list
  -rsrc xxx           synchronize resource forks (true/false/default)
  -xattrignore xxx    add a pattern to the xattrignore list
  -xattrignorenot xxx add a pattern to the xattrignorenot list
  -xattrs             synchronize extended attributes (xattrs)

 How to sync:
  -backup xxx         add a pattern to the backup list
  -backupcurr xxx     add a pattern to the backupcurr list
  -backupcurrnot xxx  add a pattern to the backupcurrnot list
  -backupdir xxx      directory for storing centralized backups
```

```
 -backuploc xxx      where backups are stored ('local' or 'central')
 -backupnot xxx      add a pattern to the backupnot list
 -backupprefix xxx   prefix for the names of backup files
 -backups            (deprecated) keep backup copies of all files (see also
                     'backup')
 -backupsuffix xxx   a suffix to be added to names of backup files
 -confirmbigdel      ask about whole-replica (or path) deletes (default true)
 -confirmmerge       ask for confirmation before committing results of a merge
 -copyonconflict     keep copies of conflicting files
 -dontchmod          when set, never use the chmod system call
 -fastcheck xxx      do fast update detection (true/false/default)
 -fat                use appropriate options for FAT filesystems
 -ignoreinodenumbers ignore inode number changes when detecting updates
 -maxbackups n       number of backed up versions of a file
 -numericids         don't map uid/gid values by user/group names
 -sortbysize         list changed files by size, not name
 -sortfirst xxx      add a pattern to the sortfirst list
 -sortlast xxx       add a pattern to the sortlast list
 -sortnewfirst       list new before changed files

How to sync (text interface (CLI) only):
 -repeat xxx         synchronize repeatedly (text interface only)
 -retry n            re-try failed synchronizations N times (text ui only)

Text interface (CLI):
 -color xxx          use color output for text UI (true/false/default)
 -dumbtty            do not change terminal settings in text UI

Graphical interface (GUI):
 -height n           height (in lines) of main window in graphical interface

Remote connections:
 -addversionno       add version number to name of unison on server
 -clientHostName xxx set host name of client
 -halfduplex         (deprecated) force half-duplex communication with the
                     server
 -killserver         kill server when done (even when using sockets)
 -listen xxx         listen on this name or addr in server socket mode (can
                     repeat)
 -rsync              activate the rsync transfer mode (default true)
 -servercmd xxx      name of unison executable on remote server
 -socket xxx         act as a server on a socket
 -sshargs xxx        other arguments (if any) for remote shell command
 -sshcmd xxx         path to the ssh executable
 -stream             (deprecated) use a streaming protocol for transferring
                     file contents (default true)
 -testserver         exit immediately after the connection to the server
 -xferbycopying      optimize transfers using local copies (default true)

Archive management:
 -ignorearchives     ignore existing archive files

Other:
```

```
   -addprefsto xxx     file to add new prefs to
   -contactquietly     suppress the 'contacting server' message during startup
   -copymax n          (deprecated) maximum number of simultaneous copyprog
                       transfers
   -copyprog xxx       (deprecated) external program for copying large files
   -copyprogrest xxx   (deprecated) variant of copyprog for resuming partial
                       transfers
   -copythreshold n    (deprecated) use copyprog on files bigger than this (if
                       >=0, in Kb)
   -diff xxx           set command for showing differences between files
   -ignorelocks        ignore locks left over from previous run (dangerous!)
   -include xxx        include a profile's preferences
   -key xxx            define a keyboard shortcut for this profile (in some UIs)
   -label xxx          provide a descriptive string label for this profile
   -log                record actions in logfile (default true)
   -logfile xxx        logfile name
   -maxerrors n        maximum number of errors before a directory transfer is
                       aborted
   -maxsizethreshold n prevent transfer of files bigger than this (if >=0, in
                       Kb)
   -maxthreads n       maximum number of simultaneous file transfers
   -mountpoint xxx     abort if this path does not exist
   -rootalias xxx      register alias for canonical root names
   -showarchive        show 'true names' (for rootalias) of roots and archive
   -source xxx         include a file's preferences
   -ui xxx             select UI ('text' or 'graphic'); command-line only
   -unicode xxx        assume Unicode encoding in case insensitive mode
   -watch              when set, use a file watcher process to detect changes

Expert options:
   -debug xxx          debug module xxx ('all' -> everything, 'verbose' -> more)
   -dumparchives       dump contents of archives just after loading
   -fastercheckUNSAFE  skip computing fingerprints for new files (experts only!)
   -selftest           run internal tests and exit
```

Here, in more detail, is what they do. Many are discussed in greater detail in other sections of the manual.

It should be noted that some command-line arguments are handled specially during startup, including `-doc`, `-help`, `-version`, `-socket`, and `-ui`. They are expected to appear on the command-line only, not in a profile. In particular, `-version` and `-doc` will print to the standard output, so they only make sense if invoked from the command-line (and not a click-launched gui that has no standard output). Furthermore, the actions associated with these command-line arguments are executed without loading a profile or doing the usual command-line parsing.

acl When this flag is set to `true`, the ACLs of files and directories are synchronized. The type of ACLs depends on the platform and filesystem support. On Unix-like platforms it can be NFSv4 ACLs, for example.

addprefsto `xxx` By default, new preferences added by Unison (e.g., new `ignore` clauses) will be appended to whatever preference file Unison was told to load at the beginning of the run. Setting the preference `addprefsto` *`filename`* makes Unison add new preferences to the file named *`filename`* instead.

addversionno When this flag is set to `true`, Unison will use `unison-`*`currentmajorversionnumber`* instead of just `unison` as the remote server command (note that the minor version number is dropped – e.g.,

unison-2.51). This allows multiple binaries for different versions of unison to coexist conveniently on the same server: whichever version is run on the client, the same version will be selected on the server.

atomic `xxx` This preference specifies paths for directories whose contents will be considered as a group rather than individually when they are both modified. The backups are also made atomically in this case. The option `backupcurr` however has no effect on atomic directories.

auto When set to `true`, this flag causes the user interface to skip asking for confirmations on non-conflicting changes. (More precisely, when the user interface is done setting the propagation direction for one entry and is about to move to the next, it will skip over all non-conflicting entries and go directly to the next conflict.)

backup `xxx` Including the preference `-backup` *`pathspec`* causes Unison to keep backup files for each path that matches *`pathspec`*; directories (nor their permissions or any other metadata) are not backed up. These backup files are kept in the directory specified by the `backuplocation` preference. The backups are named according to the `backupprefix` and `backupsuffix` preferences. The number of versions that are kept is determined by the `maxbackups` preference.

The syntax of *`pathspec`* is described in Section 6.12 [Path Specification].

backupcurr `xxx` Including the preference `-backupcurr` *`pathspec`* causes Unison to keep a backup of the *current* version of every file matching *`pathspec`*. This file will be saved as a backup with version number 000. Such backups can be used as inputs to external merging programs, for instance. See the documentation for the `merge` preference. For more details, see Section 6.8 [Merging Conflicting Versions].

The syntax of *`pathspec`* is described in Section 6.12 [Path Specification].

backupcurrnot `xxx` Exceptions to `backupcurr`, like the `ignorenot` preference.

backupdir `xxx` If this preference is set, Unison will use it as the name of the directory used to store backup files specified by the `backup` preference, when `backuplocation` is set to `central`. It is checked *after* the `UNISONBACKUPDIR` environment variable.

backuploc `xxx` This preference determines whether backups should be kept locally, near the original files, or in a central directory specified by the `backupdir` preference. If set to `local`, backups will be kept in the same directory as the original files, and if set to `central`, `backupdir` will be used instead.

backupnot `xxx` The values of this preference specify paths or individual files or regular expressions that should *not* be backed up, even if the `backup` preference selects them—i.e., it selectively overrides `backup`.

backupprefix `xxx` When a backup for a file `NAME` is created, it is stored in a directory specified by `backuplocation`, in a file called `backupprefixNAMEbackupsuffix`. `backupprefix` can include a directory name (causing Unison to keep all backup files for a given directory in a subdirectory with this name), and both `backupprefix` and `backupsuffix` can contain the string *`$VERSION`*, which will be replaced by the *age* of the backup (1 for the most recent, 2 for the second most recent, and so on...). This keyword is ignored if it appears in a directory name in the prefix; if it does not appear anywhere in the prefix or the suffix, it will be automatically placed at the beginning of the suffix.

One thing to be careful of: If the `backuploc` preference is set to `local`, Unison will automatically ignore *all* files whose prefix and suffix match `backupprefix` and `backupsuffix`. So be careful to choose values for these preferences that are sufficiently different from the names of your real files.

backups *(Deprecated)* Setting this flag to true is equivalent to setting `backuplocation` to `local` and `backup` to `Name *`.

backupsuffix `xxx` See `backupprefix` for full documentation.

batch When this is set to `true`, the user interface will ask no questions at all. Non-conflicting changes will be propagated; conflicts will be skipped.

clientHostName `xxx` When specified, the host name of the client will not be guessed and the provided host name will be used to find the archive.

color `xxx` When set to `true`, this flag enables color output in text mode user interface. When set to `false`, all color output is disabled. Default is to enable color if the `NO_COLOR` environment variable is not set.

confirmbigdel When this is set to `true`, Unison will request an extra confirmation if it appears that the entire replica has been deleted, before propagating the change. If the `batch` flag is also set, synchronization will be aborted. When the `path` preference is used, the same confirmation will be requested for top-level paths. (At the moment, this flag only affects the text user interface.) See also the `mountpoint` preference.

confirmmerge Setting this preference causes both the text and graphical interfaces to ask the user if the results of a merge command may be committed to the replica or not. Since the merge command works on temporary files, the user can then cancel all the effects of applying the merge if it turns out that the result is not satisfactory. In batch-mode, this preference has no effect. Default is false.

contactquietly If this flag is set, Unison will skip displaying the 'Contacting server' message (which some users find annoying) during startup.

copymax `n` *(Deprecated)* A number indicating how many instances of the external copying utility Unison is allowed to run simultaneously (default to 1).

copyonconflict When this flag is set, Unison will make a copy of files that would otherwise be overwritten or deleted in case of conflicting changes, and more generally whenever the default behavior is overridden. This makes it possible to automatically resolve conflicts in a fairly safe way when synchronizing continuously, in combination with the `-repeat watch` and `-prefer newer` preferences.

copyprog `xxx` *(Deprecated)* A string giving the name of an external program that can be used to copy large files efficiently (plus command-line switches telling it to copy files in-place). The default setting invokes `rsync` with appropriate options—most users should not need to change it.

copyprogrest `xxx` *(Deprecated)* A variant of `copyprog` that names an external program that should be used to continue the transfer of a large file that has already been partially transferred. Typically, `copyprogrest` will just be `copyprog` with one extra option (e.g., `--partial`, for rsync). The default setting invokes `rsync` with appropriate options—most users should not need to change it.

copythreshold `n` *(Deprecated)* A number indicating above what filesize (in kilobytes) Unison should use the external copying utility specified by `copyprog`. Specifying 0 will cause *all* copies to use the external program; a negative number will prevent any files from using it. The default is -1.

debug `xxx` This preference is used to make Unison print various sorts of information about what it is doing internally on the standard error stream. It can be used many times, each time with the name of a module for which debugging information should be printed. Possible arguments for `debug` can be found by looking for calls to `Util.debug` in the sources (using, e.g., `grep`). Setting `-debug all` causes information from *all* modules to be printed (this mode of usage is the first one to try, if you are trying to understand something that Unison seems to be doing wrong); `-debug verbose` turns on some additional debugging output from some modules (e.g., it will show exactly what bytes are being sent across the network).

diff `xxx` This preference can be used to control the name and command-line arguments of the system utility used to generate displays of file differences. The default is '`diff -u OLDER NEWER`'. If the value of this preference contains the substrings CURRENT1 and CURRENT2, these will be replaced by the names of the files to be diffed. If the value of this preference contains the substrings NEWER and OLDER, these will be replaced by the names of files to be diffed, NEWER being the most recently modified file

of the two. Without any of these substrings, the two filenames will be appended to the command. In all cases, the filenames are suitably quoted.

doc `xxx` The command-line argument `-doc` *`secname`* causes unison to display section *`secname`* of the manual on the standard output and then exit. Use `-doc all` to display the whole manual, which includes exactly the same information as the printed and HTML manuals, modulo formatting. Use `-doc topics` to obtain a list of the names of the various sections that can be printed.

dontchmod By default, Unison uses the 'chmod' system call to set the permission bits of files after it has copied them. But in some circumstances (and under some operating systems), the chmod call always fails. Setting this preference completely prevents Unison from ever calling chmod.

dumbtty When set to `true`, this flag makes the text mode user interface avoid trying to change any of the terminal settings. (Normally, Unison puts the terminal in 'raw mode', so that it can do things like overwriting the current line.) This is useful, for example, when Unison runs in a shell inside of Emacs.

When `dumbtty` is set, commands to the user interface need to be followed by a carriage return before Unison will execute them. (When it is off, Unison recognizes keystrokes as soon as they are typed.)

This preference has no effect on the graphical user interface.

dumparchives When this preference is set, Unison will create a file unison.dump on each host, containing a text summary of the archive, immediately after loading it.

fastcheck `xxx` When this preference is set to `true`, Unison will use the modification time and length of a file as a 'pseudo inode number' when scanning replicas for updates, instead of reading the full contents of every file. (This does not apply to the very first run, when Unison will always scan all files regardless of this switch). Under Windows, this may cause Unison to miss propagating an update if the modification time and length of the file are both unchanged by the update. However, Unison will never *overwrite* such an update with a change from the other replica, since it always does a safe check for updates just before propagating a change. Thus, it is reasonable to use this switch under Windows most of the time and occasionally run Unison once with `fastcheck` set to `false`, if you are worried that Unison may have overlooked an update. For backward compatibility, `yes`, `no`, and `default` can be used in place of `true`, `false`, and `auto`. See Section 6.21 [Fast Update Detection] for more information.

fastercheckUNSAFE THIS FEATURE IS STILL EXPERIMENTAL AND SHOULD BE USED WITH EXTREME CAUTION.

When this flag is set to `true`, Unison will compute a 'pseudo-fingerprint' the first time it sees a file (either because the file is new or because Unison is running for the first time). This enormously speeds update detection, but it must be used with care, as it can cause Unison to miss conflicts: If a given path in the filesystem contains files on *both* sides that Unison has not yet seen, and if those files have the same length but different contents, then Unison will not notice the presence of a conflict. If, later, one of the files is changed, the changed file will be propagated, overwriting the other.

Moreover, even when the files are initially identical, setting this flag can lead to potentially confusing behavior: if a newly created file is later touched without being modified, Unison will treat this conservatively as a potential change (since it has no record of the earlier contents) and show it as needing to be propagated to the other replica.

Most users should leave this flag off – the small time savings of not fingerprinting new files is not worth the cost in terms of safety. However, it can be very useful for power users with huge replicas that are known to be already synchronized (e.g., because one replica is a newly created duplicate of the other, or because they have previously been synchronized with Unison but Unison's archives need to be rebuilt). In such situations, it is recommended that this flag be set only for the initial run of Unison, so that new archives can be created quickly, and then turned off for normal use.

fat When this is set to `true`, Unison will use appropriate options to synchronize efficiently and without error a replica located on a FAT filesystem on a non-Windows machine: do not synchronize permissions

(`perms = 0`); never use chmod (`dontchmod = true`); treat filenames as case insensitive (`ignorecase = true`); do not attempt to synchronize symbolic links (`links = false`); ignore inode number changes when detecting updates (`ignoreinodenumbers = true`). Any of these change can be overridden by explicitly setting the corresponding preference in the profile.

follow `xxx` Including the preference `-follow` *`pathspec`* causes Unison to treat symbolic links matching *`pathspec`* as 'invisible' and behave as if the object pointed to by the link had appeared literally at this position in the replica. See Section 6.15 [Symbolic Links] for more details. The syntax of *`pathspec`* is described in Section 6.12 [Path Specification].

force `xxx` Including the preference `-force` *`root`* causes Unison to resolve all differences (including non-conflicting changes) in favor of *`root`*. This effectively changes Unison from a synchronizer into a mirroring utility.

You can also specify a unique prefix or suffix of the path of one of the roots or a unique prefix of the hostname of a remote root.

You can also specify `-force newer` (or `-force older`) to force Unison to choose the file with the later (earlier) modtime. In this case, the `-times` preference must also be enabled. If modtimes are equal in both replicas when using `newer` or `older` then this preference will have no effect (changes will be synced as if without this preference or remain unsynced in case of a conflict).

This preference is overridden by the `forcepartial` preference.

This preference should be used only if you are *sure* you know what you are doing!

forcepartial `xxx` Including the preference `forcepartial =` *`PATHSPEC`* `->` *`root`* causes Unison to resolve all differences (even non-conflicting changes) in favor of *`root`* for the files in *`PATHSPEC`* (see Section 6.12 [Path Specification] for more information). This effectively changes Unison from a synchronizer into a mirroring utility.

You can also specify a unique prefix or suffix of the path of one of the roots or a unique prefix of the hostname of a remote root.

You can also specify `forcepartial PATHSPEC -> newer` (or `forcepartial PATHSPEC -> older`) to force Unison to choose the file with the later (earlier) modtime. In this case, the `-times` preference must also be enabled. If modtimes are equal in both replicas when using `newer` or `older` then this preference will have no effect (changes will be synced as if without this preference or remain unsynced in case of a conflict).

This preference should be used only if you are *sure* you know what you are doing!

group When this flag is set to `true`, the group attributes of the files are synchronized. Whether the group names or the group identifiers are synchronized depends on the preference `numerids`.

halfduplex *(Deprecated)* When this flag is set to `true`, Unison network communication is forced to be half duplex (the client and the server never simultaneously emit data). If you experience unstabilities with your network link, this may help.

height `n` Used to set the height (in lines) of the main window in the graphical user interface.

i Provide this preference in the command line arguments to enable interactive profile manager in the text user interface. Currently only profile listing and interactive selection are available. Preferences like `batch` and `silent` remain applicable to synchronization functionality.

ignore `xxx` Including the preference `-ignore` *`pathspec`* causes Unison to completely ignore paths that match *`pathspec`* (as well as their children). This is useful for avoiding synchronizing temporary files, object files, etc. The syntax of *`pathspec`* is described in Section 6.12 [Path Specification], and further details on ignoring paths is found in Section 6.13 [Ignoring Paths].

ignorearchives When this preference is set, Unison will ignore any existing archive files and behave as though it were being run for the first time on these replicas. It is not a good idea to set this option in a profile: it is intended for command-line use.

ignorecase `xxx` When set to `true`, this flag causes Unison to treat filenames as case insensitive—i.e., files in the two replicas whose names differ in (upper- and lower-case) 'spelling' are treated as the same file. When the flag is set to `false`, Unison will treat all filenames as case sensitive. Ordinarily, when the flag is set to `default`, filenames are automatically taken to be case-insensitive if either host is running Windows or OSX. In rare circumstances it may be useful to set the flag manually.

ignoreinodenumbers When set to true, this preference makes Unison not take advantage of inode numbers during fast update detection. This switch should be used with care, as it is less safe than the standard update detection method, but it can be useful with filesystems which do not support inode numbers.

ignorelocks When this preference is set, Unison will ignore any lock files that may have been left over from a previous run of Unison that was interrupted while reading or writing archive files; by default, when Unison sees these lock files it will stop and request manual intervention. This option should be set only if you are *positive* that no other instance of Unison might be concurrently accessing the same archive files (e.g., because there was only one instance of unison running and it has just crashed or you have just killed it). It is probably not a good idea to set this option in a profile: it is intended for command-line use.

ignorenot `xxx` This preference overrides the preference `ignore`. It gives a list of patterns (in the same format as `ignore`) for paths that should definitely *not* be ignored, whether or not they happen to match one of the `ignore` patterns.

Note that the semantics of `ignore` and `ignorenot` is a little counter-intuitive. When detecting updates, Unison examines paths in depth-first order, starting from the roots of the replicas and working downwards. Before examining each path, it checks whether it matches `ignore` and does not match `ignorenot`; in this case it skips this path *and all its descendants*. This means that, if some parent of a given path matches an `ignore` pattern, then it will be skipped even if the path itself matches an `ignorenot` pattern. In particular, putting `ignore = Path *` in your profile and then using `ignorenot` to select particular paths to be synchronized will not work. Instead, you should use the `path` preference to choose particular paths to synchronize.

immutable `xxx` This preference specifies paths for directories whose immediate children are all immutable files — i.e., once a file has been created, its contents never changes. When scanning for updates, Unison does not check whether these files have been modified; this can speed update detection significantly (in particular, for mail directories).

immutablenot `xxx` This preference overrides `immutable`.

include `xxx` Include preferences from a profile. `include` *`name`* reads the profile `name` (or file `name` in the `.unison` directory if profile `name` does not exist) and includes its contents as if it was part of a profile or given directly on command line.

key `xxx` Used in a profile to define a numeric key (0-9) that can be used in the user interface to switch immediately to this profile.

killserver When set to `true`, this flag causes Unison to kill the remote server process when the synchronization is finished. This behavior is the default for `ssh` connections, so this preference is not normally needed when running over `ssh`; it is provided so that socket-mode servers can be killed off after a single run of Unison, rather than waiting to accept future connections. (Some users prefer to start a remote socket server for each run of Unison, rather than leaving one running all the time.)

label `xxx` Used in a profile to provide a descriptive string documenting its settings. (This is useful for users that switch between several profiles, especially using the 'fast switch' feature of the graphical user interface.)

links `xxx` When set to `true`, this flag causes Unison to synchronize symbolic links. When the flag is set to `false`, symbolic links will be ignored during update detection. Ordinarily, when the flag is set to `default`, symbolic links are synchronized except when one of the hosts is running Windows. On a Windows client, Unison makes an attempt to detect if symbolic links are supported and allowed by user privileges. You may have to get elevated privileges to create symbolic links. When the flag is set to `default` and symbolic links can't be synchronized then an error is produced during update detection.

listen `xxx` When acting as a server on a TCP socket, Unison will by default listen on "any" address (0.0.0.0 and [::]). This command-line argument allows to specify a different listening address and can be repeated to listen on multiple addresses. Listening address can be specified as a host name or an IP address.

log When this flag is set, Unison will log all changes to the filesystems on a file.

logfile `xxx` By default, logging messages will be appended to the file `unison.log` in your .unison directory. Set this preference if you prefer another file. It can be a path relative to your .unison directory. Sending SIGUSR1 will close the logfile; the logfile will be re-opened (and created, if needed) automatically, to allow for log rotation.

maxbackups `n` This preference specifies the number of backup versions that will be kept by unison, for each path that matches the predicate `backup`. The default is 2.

maxerrors `n` This preference controls after how many errors Unison aborts a directory transfer. Setting it to a large number allows Unison to transfer most of a directory even when some files fail to be copied. The default is 1. If the preference is set too high, Unison may take a long time to abort in case of repeated failures (for instance, when the disk is full).

maxsizethreshold `n` A number indicating above what filesize (in kilobytes) Unison should flag a conflict instead of transferring the file. This conflict remains even in the presence of force or prefer options. A negative number will allow every transfer independently of the size. The default is -1.

maxthreads `n` This preference controls how much concurrency is allowed during the transport phase. Normally, it should be set reasonably high to maximize performance, but when Unison is used over a low-bandwidth link it may be helpful to set it lower (e.g. to 1) so that Unison doesn't soak up all the available bandwidth. The default is the special value 0, which mean 20 threads when file content streaming is deactivated and 1000 threads when it is activated.

merge `xxx` This preference can be used to run a merge program which will create a new version for each of the files and the backup, with the last backup and both replicas. The syntax of *`pathspec -> cmd`* is described in Section 6.12 [Path Specification], and further details on Merging functions are present in Section 6.8 [Merging Conflicting Versions].

mountpoint `xxx` Including the preference `-mountpoint PATH` causes Unison to check, at the end of update detection, that `PATH` exists within each root, and abort if not. This can avoid synchronzing whenremovable media is not mounted. This preference can be given more than once. See Section 6.22 [Mount Points and Removable Media].

moves-experimental When this preference is set, Unison will try to avoid transferring file contents across the network, or making a local copy, by recognizing when a file or a directory has been renamed or moved to a new location. This usually allows to propagate only the rename, without transferring or copying any data. The default value is `false`.

This feature is currently experimental and may change in incompatible ways in future versions.

nocreation `xxx` Including the preference `-nocreation` *`root`* prevents Unison from performing any file creation on root *`root`*.

You can also specify a unique prefix or suffix of the path of one of the roots or a unique prefix of the hostname of a remote root.

This preference can be included twice, once for each root, if you want to prevent any creation.

nocreationpartial `xxx` Including the preference `nocreationpartial = PATHSPEC -> root` prevents Unison from performing any file creation in *PATHSPEC* on root *root* (see Section 6.12 [Path Specification] for more information). It is recommended to use `BelowPath` patterns when selecting a directory and all its contents.

nodeletion `xxx` Including the preference `-nodeletion root` prevents Unison from performing any file deletion on root *root*.

You can also specify a unique prefix or suffix of the path of one of the roots or a unique prefix of the hostname of a remote root.

This preference can be included twice, once for each root, if you want to prevent any deletion.

nodeletionpartial `xxx` Including the preference `nodeletionpartial = PATHSPEC -> root` prevents Unison from performing any file deletion in *PATHSPEC* on root *root* (see Section 6.12 [Path Specification] for more information). It is recommended to use `BelowPath` patterns when selecting a directory and all its contents.

noupdate `xxx` Including the preference `-noupdate root` prevents Unison from performing any file update or deletion on root *root*.

You can also specify a unique prefix or suffix of the path of one of the roots or a unique prefix of the hostname of a remote root.

This preference can be included twice, once for each root, if you want to prevent any update.

noupdatepartial `xxx` Including the preference `noupdatepartial = PATHSPEC -> root` prevents Unison from performing any file update or deletion in *PATHSPEC* on root *root* (see Section 6.12 [Path Specification] for more information). It is recommended to use `BelowPath` patterns when selecting a directory and all its contents.

numericids When this flag is set to `true`, groups and users are synchronized numerically, rather than by name.

The special uid 0 and the special group 0 are never mapped via user/group names even if this preference is not set.

owner When this flag is set to `true`, the owner attributes of the files are synchronized. Whether the owner names or the owner identifiers are synchronizeddepends on the preference `numerids`.

path `xxx` When no `path` preference is given, Unison will simply synchronize the two entire replicas, beginning from the given pair of roots. If one or more `path` preferences are given, then Unison will synchronize only these paths and their children. (This is useful for doing a fast sync of just one directory, for example.) Note that `path` preferences are interpreted literally—they are not regular expressions.

perms `n` The integer value of this preference is a mask indicating which permission bits should be synchronized. It is set by default to $0o1777$: all bits but the set-uid and set-gid bits are synchronised (synchronizing these latter bits can be a security hazard). If you want to synchronize all bits, you can set the value of this preference to -1. If one of the replica is on a FAT [Windows] filesystem, you should consider using the `fat` preference instead of this preference. If you need Unison not to set permissions at all, set the value of this preference to 0 and set the preference `dontchmod` to `true`.

prefer `xxx` Including the preference `-prefer root` causes Unison always to resolve conflicts in favor of *root*, rather than asking for guidance from the user, except for paths marked by the preference `merge`. (The syntax of *root* is the same as for the `root` preference, plus the special values `newer` and `older`.)

You can also specify a unique prefix or suffix of the path of one of the roots or a unique prefix of the hostname of a remote root.

This preference is overridden by the `preferpartial` preference.

This preference should be used only if you are *sure* you know what you are doing!

preferpartial `xxx` Including the preference `preferpartial = PATHSPEC -> root` causes Unison always to resolve conflicts in favor of *root*, rather than asking for guidance from the user, for the files in *PATHSPEC* (see Section 6.12 [Path Specification] for more information). (The syntax of *root* is the same as for the `root` preference, plus the special values `newer` and `older`.)

You can also specify a unique prefix or suffix of the path of one of the roots or a unique prefix of the hostname of a remote root.

This preference should be used only if you are *sure* you know what you are doing!

repeat `xxx` Setting this preference causes the text-mode interface to synchronize repeatedly, rather than doing it just once and stopping. If the argument is a number, Unison will pause for that many seconds before beginning again. When the argument is `watch`, Unison relies on an external file monitoring process to synchronize whenever a change happens. You can combine the two with a `+` character to use file monitoring and also do a full scan every specified number of seconds. For example, `watch+3600` will react to changes immediately and additionally do a full scan every hour.

retry `n` Setting this preference causes the text-mode interface to try again to synchronize updated paths where synchronization fails. Each such path will be tried N times.

root `xxx` Each use of this preference names the root of one of the replicas for Unison to synchronize. Exactly two roots are needed, so normal modes of usage are either to give two values for `root` in the profile, or to give no values in the profile and provide two on the command line. Details of the syntax of roots can be found in Section 5.1 [Roots].

The two roots can be given in either order; Unison will sort them into a canonical order before doing anything else. It also tries to 'canonize' the machine names and paths that appear in the roots, so that, if Unison is invoked later with a slightly different name for the same root, it will be able to locate the correct archives.

rootalias `xxx` When calculating the name of the archive files for a given pair of roots, Unison replaces any roots matching the left-hand side of any rootalias rule by the corresponding right-hand side.

rsrc `xxx` When set to `true`, this flag causes Unison to synchronize resource forks and HFS meta-data. On filesystems that do not natively support resource forks, this data is stored in Carbon-compatible ._ AppleDouble files. When the flag is set to `false`, Unison will not synchronize these data. Ordinarily, the flag is set to `default`, and these data are automatically synchronized if either host is running OSX. In rare circumstances it is useful to set the flag manually.

rsync Unison uses the 'rsync algorithm' for 'diffs-only' transfer of updates to large files. Setting this flag to false makes Unison use whole-file transfers instead. Under normal circumstances, there is no reason to do this, but if you are having trouble with repeated 'rsync failure' errors, setting it to false should permit you to synchronize the offending files.

selftest Run internal tests and exit. This option is mostly for developers and must be used carefully: in particular, it will delete the contents of both roots, so that it can install its own files for testing. This flag only makes sense on the command line. When it is provided, no preference file is read: all preferences must be specified on thecommand line. Also, since the self-test procedure involves overwriting the roots and backup directory, the names of the roots and of the backupdir preference must include the string "test" or else the tests will be aborted. (If these are not given on the command line, dummy subdirectories in the current directory will be created automatically.)

servercmd `xxx` This preference can be used to explicitly set the name of the Unison executable on the remote server (e.g., giving a full path name), if necessary.

showarchive When this preference is set, Unison will print out the 'true names'of the roots, in the same form as is expected by the `rootalias` preference.

silent When this preference is set to `true`, the textual user interface will print nothing at all, except in the case of errors. Setting `silent` to true automatically sets the `batch` preference to `true`.

socket `xxx` Start unison as a server listening on a TCP socket (with TCP port number as argument) or a local socket (aka Unix domain socket) (with socket path as argument).

sortbysize When this flag is set, the user interface will list changed files by size (smallest first) rather than by name. This is useful, for example, for synchronizing over slow links, since it puts very large files at the end of the list where they will not prevent smaller files from being transferred quickly.

This preference (as well as the other sorting flags, but not the sorting preferences that require patterns as arguments) can be set interactively and temporarily using the 'Sort' menu in the graphical and text user interfaces.

sortfirst `xxx` Each argument to `sortfirst` is a pattern *`pathspec`*, which describes a set of paths. Files matching any of these patterns will be listed first in the user interface. The syntax of *`pathspec`* is described in Section 6.12 [Path Specification].

sortlast `xxx` Similar to `sortfirst`, except that files matching one of these patterns will be listed at the very end.

sortnewfirst When this flag is set, the user interface will list newly created files before all others. This is useful, for example, for checking that newly created files are not 'junk', i.e., ones that should be ignored or deleted rather than synchronized.

source `xxx` Include preferences from a file. `source` *`name`* reads the file `name` in the `.unison` directory and includes its contents as if it was part of a profile or given directly on command line.

sshargs `xxx` The string value of this preference will be passed as additional arguments (besides the host name and the name of the Unison executable on the remote system) to the `ssh` command used to invoke the remote server. The backslash is an escape character.

sshcmd `xxx` This preference can be used to explicitly set the name of the ssh executable (e.g., giving a full path name), if necessary.

stream *(Deprecated)* When this preference is set, Unison will use an experimental streaming protocol for transferring file contents more efficiently. The default value is `true`.

terse When this preference is set to `true`, the user interface will not print status messages.

testserver Setting this flag on the command line causes Unison to attempt to connect to the remote server and, if successful, print a message and immediately exit. Useful for debugging installation problems. Should not be set in preference files.

times When this flag is set to `true`, file modification times (but not directory modtimes) are propagated.

ui `xxx` This preference selects either the graphical or the textual user interface. Legal values are `graphic` or `text`.

Because this option is processed specially during Unison's start-up sequence, it can *only* be used on the command line. In preference files it has no effect.

If the Unison executable was compiled with only a textual interface, this option has no effect. (The pre-compiled binaries are all compiled with both interfaces available.)

unicode `xxx` When set to `true`, this flag causes Unison to perform case insensitive file comparisons assuming Unicode encoding. This is the default. When the flag is set to `false`, Latin 1 encoding is assumed (this means that all bytes that are not letters in Latin 1 encoding will be compared byte-for-byte, even if they may be valid characters in some other encoding). When Unison runs in case sensitive mode, this flag only makes a difference if one host is running Mac OS X. Under Mac OS X, it selects whether comparing the filenames up to decomposition, or byte-for-byte.

version Print the current version number and exit. (This option only makes sense on the command line.)

watch Unison uses a file watcher process, when available, to detect filesystem changes; this is used to speed up update detection. Setting this flag to false disables the use of this process.

xattrignore `xxx` Preference `-xattrignore` *`namespec`* causes Unison to ignore extended attributes with names that match *`namespec`*. This can be used to exclude extended attributes that would fail synchronization due to lack of permissions or technical differences at replicas. The syntax of *`namespec`* is the same as used for path specification (described in Section 6.12 [Path Specification]); prefer the `Path` and `Regex` forms over the `Name` form. The pattern is applied to the *name* of extended attribute, not to path. *On Linux*, attributes in the security and trusted namespaces are ignored by default (this is achieved by pattern `Regex !(security|trusted)[.].*`); also attributes used to store POSIX ACL are ignored by default (this is achieved by pattern `Path !system.posix_acl_*`). To sync attributes in one or both of these namespaces, see the `xattrignorenot` preference. Note that the namespace name must be prefixed with a "!" (applies on Linux only). All names not prefixed with a "!" are taken as strictly belonging to the user namespace and therefore the "!user." prefix is never used.

xattrignorenot `xxx` This preference overrides the preference `xattrignore`. It gives a list of patterns (in the same format as `xattrignore`) for extended attributes that should *not* be ignored, whether or not they happen to match one of the `xattrignore` patterns. It is possible to synchronize only desired attributes by ignoring all attributes (for example, by setting `xattrignore` to `Path *` and then adding `xattrignorenot` for extended attributes that should be synchronized. *On Linux*, attributes in the security and trusted namespaces are ignored by default. To sync attributes in one or both of these namespaces, you may add an `xattrignorenot` pattern like `Path !security.*` to sync all attributes in the security namespace, or `Path !security.selinux` to sync a specific attribute in an otherwise ignored namespace. A pattern like `Path !system.posix_acl_*` can be used to sync POSIX ACLs on Linux. Note that the namespace name must be prefixed with a "!" (applies on Linux only). All names not prefixed with a "!" are taken as strictly belonging to the user namespace and therefore the "!user." prefix is never used.

xattrs When this flag is set to `true`, the extended attributes of files and directories are synchronized. System extended attributes are not synchronized.

xferbycopying When this preference is set, Unison will try to avoid transferring file contents across the network by recognizing when a file with the required contents already exists in the target replica. This usually allows file moves to be propagated very quickly. The default value is `true`.

6.5 Profiles

A *profile* is a text file that specifies permanent settings for roots, paths, ignore patterns, and other preferences, so that they do not need to be typed at the command line every time Unison is run. Profiles should reside in the `.unison` directory on the client machine. If Unison is started with just one argument *`name`* on the command line, it looks for a profile called *`name`*`.prf` in the `.unison` directory. If it is started with no arguments, it scans the `.unison` directory for files whose names end in `.prf` and offers a menu (provided that the Unison executable is compiled with the graphical user interface). If a file named `default.prf` is found, its settings will be offered as the default choices.

To set the value of a preference `p` permanently, add to the appropriate profile a line of the form

```
p = true
```

for a boolean flag or

```
p = <value>
```

for a preference of any other type.

A profile may include blank lines and lines beginning with `#`; both are ignored.

Spaces and tabs before and after `p` and `xxx` are ignored. Spaces, tabs, and non-printable characters within values are not treated specially, so that e.g. `root = /foo bar` refers to a directory containing a space. (On

systems using newline for line ending, carriage returns are currently ignored, but this is not part of the specification.)

When Unison starts, it first reads the profile and then the command line, so command-line options will override settings from the profile.

Profiles may also include lines of the form `include` *name*, which will cause the file *name* (or *name*`.prf`, if *name* does not exist in the `.unison` directory) to be read at the point, and included as if its contents, instead of the `include` line, was part of the profile. Include lines allows settings common to several profiles to be stored in one place. A similar line of the form `source` *name* does the same except that it does not attempt to add a suffix to *name*. Similar lines of the form `include?` *name* or `source?` *name* do the same as their respective lines without the question mark except that it does not constitute an error to specify a non-existing file *name*. In *name* the backslash is an escape character.

A profile may include a preference '`label =` *desc*' to provide a description of the options selected in this profile. The string *desc* is listed along with the profile name in the profile selection dialog, and displayed in the top-right corner of the main Unison window in the graphical user interface.

The graphical user-interface also supports one-key shortcuts for commonly used profiles. If a profile contains a preference of the form '`key =` *n*', where *n* is a single digit, then pressing this digit key will cause Unison to immediately switch to this profile and begin synchronization again from scratch. In this case, all actions that have been selected for a set of changes currently being displayed will be discarded.

6.6 Sample Profiles

6.6.1 A Minimal Profile

Here is a very minimal profile file, such as might be found in `.unison/default.prf`:

```
# Roots of the synchronization
root = /home/bcpierce
root = ssh://saul//home/bcpierce

# Paths to synchronize
path = current
path = common
path = .netscape/bookmarks.html
```

6.6.2 A Basic Profile

Here is a more sophisticated profile, illustrating some other useful features.

```
# Roots of the synchronization
root = /home/bcpierce
root = ssh://saul//home/bcpierce

# Paths to synchronize
path = current
path = common
path = .netscape/bookmarks.html

# Some regexps specifying names and paths to ignore
ignore = Name temp.*
ignore = Name *~
ignore = Name .*~
ignore = Path */pilot/backup/Archive_*
ignore = Name *.o
ignore = Name *.tmp
```

```
# Window height
height = 37

# Keep a backup copy of every file in a central location
backuplocation = central
backupdir = /home/bcpierce/backups
backup = Name *
backupprefix = $VERSION.
backupsuffix =

# Use this command for displaying diffs
diff = diff -y -W 79 --suppress-common-lines

# Log actions to the terminal
log = true
```

6.6.3 A Power-User Profile

When Unison is used with large replicas, it is often convenient to be able to synchronize just a part of the replicas on a given run (this saves the time of detecting updates in the other parts). This can be accomplished by splitting up the profile into several parts — a common part containing most of the preference settings, plus one "top-level" file for each set of paths that need to be synchronized. (The `include` mechanism can also be used to allow the same set of preference settings to be used with different roots.)

The collection of profiles implementing this scheme might look as follows. The file `default.prf` is empty except for an `include` directive:

```
# Include the contents of the file common
include common
```

Note that the name of the common file is `common`, not `common.prf`; this prevents Unison from offering `common` as one of the list of profiles in the opening dialog (in the graphical UI).

The file `common` contains the real preferences:

```
# Roots of the synchronization
root = /home/bcpierce
root = ssh://saul//home/bcpierce

# (... other preferences ...)

# If any new preferences are added by Unison (e.g. 'ignore'
# preferences added via the graphical UI), then store them in the
# file 'common' rather than in the top-level preference file
addprefsto = common

# Names and paths to ignore:
ignore = Name temp.*
ignore = Name *~
ignore = Name .*~
ignore = Path */pilot/backup/Archive_*
ignore = Name *.o
ignore = Name *.tmp
```

Note that there are no `path` preferences in `common`. This means that, when we invoke Unison with the default profile (e.g., by typing '`unison default`' or just '`unison`' on the command line), the whole replicas will be synchronized. (If we *never* want to synchronize the whole replicas, then `default.prf` would instead include settings for all the paths that are usually synchronized.)

To synchronize just part of the replicas, Unison is invoked with an alternate preference file—e.g., doing '`unison workingset`', where the preference file `workingset.prf` contains

```
path = current/papers
path = Mail/inbox
path = Mail/drafts
include common
```

causes Unison to synchronize just the listed subdirectories.

The `key` preference can be used in combination with the graphical UI to quickly switch between different sets of paths. For example, if the file `mail.prf` contains

```
path = Mail
batch = true
key = 2
include common
```

then pressing 2 will cause Unison to look for updates in the `Mail` subdirectory and (because the `batch` flag is set) immediately propagate any that it finds.

6.7 Keeping Backups

When Unison overwrites (or deletes) a file or directory while propagating changes from the other replica, it can keep the old version around as a backup. There are several preferences that control precisely where these backups are stored and how they are named.

To enable backups, you must give one or more `backup` preferences. Each of these has the form

```
backup = <pathspec>
```

where `<pathspec>` has the same form as for the `ignore` preference. For example,

```
backup = Name *
```

causes Unison to create backups of *all* files and directories. The `backupnot` preference can be used to give a few exceptions: it specifies which files and directories should *not* be backed up, even if they match the `backup` pathspec.

It is important to note that the `pathspec` is matched against the path that is being updated by Unison, not its descendants. For example, if you set `backup = Name *.txt` and then delete a whole directory named `foo` containing some text files, these files will not be backed up because Unison will just check that `foo` does not match `*.txt`. Similarly, if the directory itself happened to be called `foo.txt`, then the whole directory and all the files in it will be backed up, regardless of their names.

Backup files can be stored either *centrally* or *locally*. This behavior is controlled by the preference `backuplocation`, whose value must be either `central` or `local`. (The default is `central`.) Note that central storage of backups can lead to backup files being stored in a different filesystem than the original files, which could have different security properties and different amounts of available storage.

When backups are stored locally, they are kept in the same directory as the original.

When backups are stored centrally, the directory used to hold them is controlled by the preference `backupdir` and the environment variable `UNISONBACKUPDIR`. (The environment variable is checked first.) If neither of these are set, then the directory `.unison/backup` in the user's home directory is used.

The preference `maxbackups` (default 2) controls how many previous versions of each file are kept (including the current version), following the usual plan of deleting the oldest when creating a new one.

By default, backup files are named `.bak.VERSION.FILENAME`, where `FILENAME` is the original filename and `VERSION` is the backup number (1 for the most recent, 2 for the next most recent, etc.). This can be changed by setting the preferences `backupprefix` and/or `backupsuffix`. If desired, `backupprefix` may include a directory prefix; this can be used with `backuplocation = local` to put all backup files for each directory into a single subdirectory. For example, setting

```
backuplocation = local
backupprefix = .unison/$VERSION.
backupsuffix =
```

will put all backups in a local subdirectory named `.unison`. Also, note that the string `$VERSION` in either `backupprefix` or `backupsuffix` (it must appear in one or the other) is replaced by the version number. This can be used, for example, to ensure that backup files retain the same extension as the originals.

Other than `maxbackups` (which will never delete the last backup), there are no other mechanisms for deleting backups.

For backward compatibility, the `backups` preference is also supported. It simply means `backup = Name *` and `backuplocation = local`.

6.8 Merging Conflicting Versions

Unison can invoke external programs to merge conflicting versions of a file. The preference `merge` controls this process.

The `merge` preference may be given once or several times in a preference file (it can also be given on the command line, of course, but this tends to be awkward because of the spaces and special characters involved). Each instance of the preference looks like this:

```
merge = <PATHSPEC> -> <MERGECMD>
```

The `<PATHSPEC>` here has exactly the same format as for the `ignore` preference (see Section 6.12 [Path Specification]). For example, using "`Name *.txt`" as the `<PATHSPEC>` tells Unison that this command should be used whenever a file with extension `.txt` needs to be merged.

Many external merging programs require as inputs not just the two files that need to be merged, but also a file containing the *last synchronized version*. You can ask Unison to keep a copy of the last synchronized version for some files using the `backupcurrent` preference. This preference is used in exactly the same way as `backup` and its meaning is similar, except that it causes backups to be created of the *current* contents of each file after it has been synchronized by Unison, rather than the *previous* contents that Unison overwrote. These backups are stored in *both* replicas in the same place as ordinary backup files—i.e. according to the `backuplocation` and `backupdir` preferences. They are named like the original files if `backupslocation` is set to 'central' and otherwise, Unison uses the `backupprefix` and `backupsuffix` preferences and assumes a version number 000 for these backups. Note that there are no mechanisms (beyond the limit on the number of backups for each file) to remove backup files.

The `<MERGECMD>` part of the preference specifies what external command should be invoked to merge files at paths matching the `<PATHSPEC>`. Within this string, several special substrings are recognized; these will be substituted with appropriate values before invoking a sub-shell to execute the command.

- `CURRENT1` is replaced by the name of (a temporary copy of) the local variant of the file.
- `CURRENT2` is replaced by the name of a temporary file, into which the contents of the remote variant of the file have been transferred by Unison prior to performing the merge.
- `CURRENTARCH` is replaced by the name of the backed up copy of the original version of the file (i.e., the file saved by Unison if the current filename matches the path specifications for the `backupcurrent` preference, as explained above), if one exists. If no archive exists and `CURRENTARCH` appears in the merge command, then an error is signalled.
- `CURRENTARCHOPT` is replaced by the name of the backed up copy of the original version of the file (i.e., its state at the end of the last successful run of Unison), if one exists, or the empty string if no archive exists.
- `NEW` is replaced by the name of a temporary file that Unison expects to be written by the merge program when it finishes, giving the desired new contents of the file.
- `PATH` is replaced by the path (relative to the roots of the replicas) of the file being merged.

- NEW1 and NEW2 are replaced by the names of temporary files that Unison expects to be written by the merge program when it is only able to partially merge the originals; in this case, NEW1 will be written back to the local replica and NEW2 to the remote replica; NEWARCH, if present, will be used as the "last common state" of the replicas. (These three options are provided for later compatibility with the Harmony data synchronizer.)

- BATCHMODE is replaced according to the batch mode of Unison; if it is in batch mode, then a non empty string ("batch") is substituted, otherwise the empty string is substituted.

To accommodate the wide variety of programs that users might want to use for merging, Unison checks for several possible situations when the merge program exits:

- If the merge program exits with a non-zero status, then merge is considered to have failed and the replicas are not changed.

- If the file NEW has been created, it is written back to both replicas (and stored in the backup directory). Similarly, if just the file NEW1 has been created, it is written back to both replicas.

- If neither NEW nor NEW1 have been created, then Unison examines the temporary files CURRENT1 and CURRENT2 that were given as inputs to the merge program. If either has been changed (or both have been changed in identical ways), then its new contents are written back to both replicas. If either CURRENT1 or CURRENT2 has been *deleted*, then the contents of the other are written back to both replicas.

- If the files NEW1, NEW2, and NEWARCH have all been created, they are written back to the local replica, remote replica, and backup directory, respectively. If the files NEW1, NEW2 have been created, but NEWARCH has not, then these files are written back to the local replica and remote replica, respectively. Also, if NEW1 and NEW2 have identical contents, then the same contents are stored as a backup (if the backupcurrent preference is set for this path) to reflect the fact that the path is currently in sync.

- If NEW1 and NEW2 (resp. CURRENT1 and CURRENT2) are created (resp. overwritten) with different contents but the merge command did not fail (i.e., it exited with status code 0), then we copy NEW1 (resp. CURRENT1) to the other replica and to the archive.

  This behavior is a design choice made to handle the case where a merge command only synchronizes some specific contents between two files, skipping some irrelevant information (order between entries, for instance). We assume that, if the merge command exits normally, then the two resulting files are "as good as equal." (The reason we copy one on top of the other is to avoid Unison detecting that the files are unequal the next time it is run and trying again to merge them when, in fact, the merge program has already made them as similar as it is able to.)

You can disable a merge by setting a <MERGECMD> that does nothing. For example you can override the merging of text files specified in a profile by typing on the command line:

```
unison profile -merge 'Name *.txt -> echo SKIP'
```

If the confirmmerge preference is set and Unison is not run in batch mode, then Unison will always ask for confirmation before actually committing the results of the merge to the replicas.

You can detect batch mode by testing BATCHMODE; for example to avoid a merge completely do nothing:

```
merge = Name *.txt -> [ -z "BATCHMODE" ] && mergecmd CURRENT1 CURRENT2
```

A large number of external merging programs are available. For example, on Unix systems setting the merge preference to

```
merge = Name *.txt -> diff3 -m CURRENT1 CURRENTARCH CURRENT2
                        > NEW || echo "differences detected"
```

will tell Unison to use the external diff3 program for merging. Alternatively, users of emacs may find the following settings convenient:

```
merge = Name *.txt -> emacs -q --eval '(ediff-merge-files-with-ancestor
                          "CURRENT1" "CURRENT2" "CURRENTARCH" nil "NEW")'
```

(These commands are displayed here on two lines to avoid running off the edge of the page. In your preference file, each command should be written on a single line.)

Users running emacs under windows may find something like this useful:

```
merge = Name * -> C:\Progra~1\Emacs\emacs\bin\emacs.exe -q --eval
                        "(ediff-files """CURRENT1""" """CURRENT2""")"
```

Users running Mac OS X (you may need the Developer Tools installed to get the `opendiff` utility) may prefer

```
merge = Name *.txt -> opendiff CURRENT1 CURRENT2 -ancestor CURRENTARCH -merge NEW
```

Here is a slightly more involved hack. The `opendiff` program can operate either with or without an archive file. A merge command of this form

```
merge = Name *.txt ->
          if [ CURRENTARCHOPTx = x ];
          then opendiff CURRENT1 CURRENT2 -merge NEW;
          else opendiff CURRENT1 CURRENT2 -ancestor CURRENTARCHOPT -merge NEW;
          fi
```

(still all on one line in the preference file!) will test whether an archive file exists and use the appropriate variant of the arguments to `opendiff`.

Linux users may enjoy this variant:

```
merge = Name * -> kdiff3 -o NEW CURRENTARCHOPT CURRENT1 CURRENT2
```

Ordinarily, external merge programs are only invoked when Unison is *not* running in batch mode. To specify an external merge program that should be used no matter the setting of the `batch` flag, use the `mergebatch` preference instead of `merge`.

> *Please post suggestions for other useful values of the* `merge` *preference to the* `unison-users` *mailing list—we'd like to give several examples here.*

6.9 The User Interface

Both the textual and the graphical user interfaces are intended to be mostly self-explanatory. Here are just a few tricks:

- By default, when running on Unix the textual user interface will try to put the terminal into the "raw mode" so that it reads the input a character at a time rather than a line at a time. (This means you can type just the single keystroke "`>`" to tell Unison to propagate a file from left to right, rather than "`>` Enter.")

  There are some situations, though, where this will not work — for example, when Unison is running in a shell window inside Emacs. Setting the `dumbtty` preference will force Unison to leave the terminal alone and process input a line at a time.

6.10 Interrupting a Synchronization

It is possible to interrupt an ongoing synchronization process before it completes. Different user interfaces offer different ways of doing it.

Graphical Interface:

In the graphical user interface the synchronization process can be interrupted before it is finished by pressing the "Stop" button or by closing the window. The "Stop" button causes the onging propagation to be stopped as quickly as possible while still doing proper cleanup. The application keeps running and a rescan can be performed or a different profile selected. Closing the window in the middle of update propagation process will exit the application immediately without doing proper cleanup; it is therefore not recommended unless the "Stop" button does not react quickly enough.

Textual Interface:

When not synchronizing continuously, the text interface terminates when synchronization is finished normally or due to a fatal error occurring.

In the text interface, to interrupt synchronization before it is finished, press "Ctrl-C" (or send signal `SIGINT` or `SIGTERM`). This will interrupt update propagation as quickly as possible but still complete proper cleanup. If the process does not stop even after pressing "Ctrl-C" then keep doing it repeatedly. This will bypass cleanup procedures and terminates the process forcibly (similar to `SIGKILL`). Doing so may leave the archives or replicas in an inconsistent state or locked.

When synchronizing continuously (time interval repeat or with filesystem monitoring), interrupting with "Ctrl-C" or with signal `SIGINT` or `SIGTERM` works the same way as described above and will additionally stop the continuous process. To stop only the continuous process and let the last synchronization complete normally, send signal `SIGUSR2` instead.

6.11 Exit Code

When running in the textual mode, Unison returns an exit status, which describes whether, and at which level, the synchronization was successful. The exit status could be useful when Unison is invoked from a script. Currently, there are four possible values for the exit status:

0 : successful synchronization; everything is up-to-date now.

1 : some files were skipped, but all file transfers were successful.

2 : non-fatal failures occurred during file transfer.

3 : a fatal error occurred, or the execution was interrupted.

The graphical interface does not return any useful information through the exit status.

6.12 Path Specification

Several Unison preferences (e.g., `ignore`/`ignorenot`, `follow`, `sortfirst`/`sortlast`, `backup`, `merge`, etc.) specify individual paths or sets of paths. These preferences share a common syntax based on regular-expressions. Each preference is associated with a list of path patterns; the paths specified are those that match any one of the path pattern.

- Pattern preferences can be given on the command line, or, more often, stored in profiles, using the same syntax as other preferences. For example, a profile line of the form

  `ignore = `*`pattern`*

  adds *`pattern`* to the list of patterns to be ignored.

- Each *`pattern`* can have one of three forms. The most general form is a POSIX Extended Regular Expression introduced by the keyword `Regex`. (The collating symbol, equivalence class expression, and character class expression described in Section 9.3.5 of the POSIX specification are not currently supported).

  `Regex `*`regexp`*

For convenience, three other styles of pattern are also recognized:

```
Name name
```

matches any path in which the last component matches *name*,

```
Path path
```

matches exactly the path *path*, and

```
BelowPath path
```

matches the path *path* and any path below. The *name* and *path* arguments of the latter forms of patterns are *not* regular expressions. Instead, standard "globbing" conventions can be used in *name* and *path*:

- a `*` matches any sequence of characters not including `/` (and not beginning with `.`, when used at the beginning of a *name*)
- a `?` matches any single character except `/` (and leading `.`)
- `[xyz]` matches any character from the set {`x`, `y`, `z`}
- `{a,bb,ccc}` matches any one of `a`, `bb`, or `ccc`. (Be careful not to put extra spaces after the commas: these will be interpreted literally as part of the strings to be matched!)

- The path separator in path patterns is always the forward-slash character "/" — even when the client or server is running under Windows, where the normal separator character is a backslash. This makes it possible to use the same set of path patterns for both Unix and Windows file systems.
- A path specification may be followed by the separator " `->` " itself followed by a string which will be associated to the matching paths:

  ```
  Path path -> associated string
  ```

  Not all pathspec preferences use these associated strings but all pathspec preferences are parsed identically and the strings may be ignored. Only the last match of the separator string on the line is used as a delimiter. Thus to allow a path specification to contain the separator string, append an associated string to it, even if it is not used. The associated string cannot contain the separator string.

Some examples of path patterns appear in Section 6.13 [Ignoring Paths]. Associated strings are used by the preference `merge`.

6.13 Ignoring Paths

Most users of Unison will find that their replicas contain lots of files that they don't ever want to synchronize — temporary files, very large files, old stuff, architecture-specific binaries, etc. They can instruct Unison to ignore these paths using patterns introduced in Section 6.12 [Path Specification].

For example, the following pattern will make Unison ignore any path containing the name `CVS` or a name ending in `.cmo`:

```
ignore = Name {CVS,*.cmo}
```

The next pattern makes Unison ignore the path `a/b`:

```
ignore = Path a/b
```

Path patterns do *not* skip filenames beginning with `.` (as Name patterns do). For example,

```
ignore = Path */tmp
```

will include `.foo/tmp` in the set of ignore directories, as it is a path, not a name, that is ignored.

The following pattern makes Unison ignore any path beginning with `a/b` and ending with a name ending by `.ml`.

```
ignore = Regex a/b/.*\.ml
```

Note that regular expression patterns are "anchored": they must match the whole path, not just a substring of the path.

Here are a few extra points regarding the `ignore` preference.

- If a directory is ignored, all its descendants will be too.
- The user interface provides some convenient commands for adding new patterns to be ignored. To ignore a particular file, select it and press "`i`". To ignore all files with the same extension, select it and press "`E`" (with the shift key). To ignore all files with the same name, no matter what directory they appear in, select it and press "`N`". These new patterns become permanent: they are immediately added to the current profile on disk.
- If you use the `include` directive to include a common collection of preferences in several top-level preference files, you will probably also want to set the `addprefsto` preference to the name of this file. This will cause any new ignore patterns that you add from inside Unison to be appended to this file, instead of whichever top-level preference file you started Unison with.
- Ignore patterns can also be specified on the command line, if you like (this is probably not very useful), using an option like `-ignore 'Name temp.txt'`.
- Be careful about renaming directories containing ignored files. Because Unison understands the rename as a delete plus a create, any ignored files in the directory will be lost (since they are invisible to Unison and therefore they do not get recreated in the new version of the directory).
- There is also an `ignorenot` preference, which specifies a set of patterns for paths that should *not* be ignored, even if they match an `ignore` pattern. However, the interaction of these two sets of patterns can be a little tricky. Here is exactly how it works:
  - Unison starts detecting updates from the root of the replicas—i.e., from the empty path. If the empty path matches an `ignore` pattern and does not match an `ignorenot` pattern, then the whole replica will be ignored. (For this reason, it is not a good idea to include `Name *` as an `ignore` pattern. If you want to ignore everything except a certain set of files, use `Name ?*`.)
  - If the root is a directory, Unison continues looking for updates in all the immediate children of the root. Again, if the name of some child matches an `ignore` pattern and does not match an `ignorenot` pattern, then this whole path *including everything below it* will be ignored.
  - If any of the non-ignored children are directories, then the process continues recursively.

6.14 Moved or Renamed Paths

Unison can, under certain conditions, detect moved and/or renamed files and directories. In that case, the move/rename is atomically propagated to the other replica, completely bypassing copying any data over the network or locally. To ask Unison to detect moves/renames, enable the `moves-experimental` preference. When the `moves-experimental` preference is not enabled then all moved and/or renamed files and directories will be detected and propagated as a separate creation at the new path and deletion at the old path, potentially having to copy a large amount of data.

It is not possible to detect actual moves and renames simply by scanning the file system. Thus, moved and/or renamed files and directories are detected heuristically by matching detected deletions and creations that have the same contents; this may detect supposed moves/renames that do not match the actions done by user. For directories, contents means the names and contents of all children recursively.

When having the same contents, a deletion and a creation are matched as a possible move/rename by following tests (up to first passing test):

- (for files only) inodes match;
- modification time and parent are same, but names are not (a renamed file/directory);
- modification time and name are the same, but parents are not (a moved file/directory);
- parent is the same, but names are not (a renamed file/directory);
- name is the same, but parents are not (a moved file/directory);
- modification time is the same, but parents and names are not (renamed and moved file/directory);
- nothing is the same (except contents) (renamed and moved file/directory).

If the contents have changed since the last sync then it will not be detected as a move/rename. If propagating a move/rename fails in the other replica then Unison falls back to a regular copy.

6.15 Symbolic Links

Ordinarily, Unison treats symbolic links in Unix replicas as "opaque": it considers the contents of the link to be just the string specifying where the link points, and it will propagate changes in this string to the other replica.

It is sometimes useful to treat a symbolic link "transparently," acting as though whatever it points to were physically *in* the replica at the point where the symbolic link appears. To tell Unison to treat a link in this manner, add a line of the form

```
follow = pathspec
```

to the profile, where *`pathspec`* is a path pattern as described in Section 6.12 [Path Specification].

> **Warning:** Be careful when using the `follow` preference. Using a `Pathspec` that is not detailed and accurate enough will cause Unison to follow symlinks that you may have not intended to. This can cause paths outside the replica to be overwritten and deleted due to updates in the other replica. This can also cause the targets of links pointing to within the replica to be synchronized under two or more names (once directly and once via the link, for example), leading to unintended results and conflicts.

Warning: Deleting, in one replica, the path where a followed symbolic link is in the other replica will cause the *target* of the "transparent" link to be deleted in the other replica. The symbolic link itself will not be deleted and remains as a broken link. This happens even if there was a symbolic link in both replicas and only the link, not the link target, was deleted in one.

Treating symbolic links "transparently" may not always work as expected when it comes to directories. Deleting a file or directory in one replica will cause the target of a "transparent" link to be deleted in the other replica. Deleting a parent directory, however, which itself might or might not be a followed link, will not delete the targets of any "transparent" links contained within.

Renaming or moving a file or directory in one replica behaves as a combination of a delete on the old path and a recreate on the new path. Thus, all the caveats regarding deletion of followed symbolic links apply (see above). Additionally, the link *target* is recreated in-place on the new path, *not* the symbolic link.

Followed symbolic links are treated "transparently" also when it comes to the `copyonconflict` and `backup` preferences. This means that the target of the link is copied to the conflict or backup location, and you need to restore to the original link target location outside the replica, should you need the backup.

Not all Windows versions and file systems support symbolic links; Unison will refuse to propagate an opaque symbolic link from Unix to Windows and flag the path as erroneous if the support or privileges are lacking on the Windows side. When a Unix replica is to be synchronized with such Windows system,

all symbolic links should match either an `ignore` pattern or a `follow` pattern. To completely ignore all symbolic links, you may set the preference `links` to `false`.

Warning: Just like with `ignore`, be careful with "`links = false`". This makes Unison effectively ignore symbolic links, so they could be deleted without notice.

You may need to acquire extra privileges to create symbolic links under Windows. By default, this is only allowed for administrators. Unison may not be able to automatically detect support for symbolic links under Windows. In that case, set the preference `links` to `true` explicitly.

6.16 Permissions

Synchronizing the permission bits of files is slightly tricky when two different filesystems are involved (e.g., when synchronizing a Windows client and a Unix server). In detail, here's how it works:

- When the permission bits of an existing file or directory are changed, the values of those bits that make sense on *both* operating systems will be propagated to the other replica. The other bits will not be changed.
- When a newly created file is propagated to a remote replica, the permission bits that make sense in both operating systems are also propagated. The values of the other bits are set to default values (they are taken from the current umask, if the receiving host is a Unix system).
- For security reasons, the Unix `setuid` and `setgid` bits are not propagated.
- The Unix owner and group ids can be propagated (see `owner` and `group` preferences) by mapping names or by numeric ids (see `numericids` preference).

6.17 Access Control Lists - ACLs

Unison allows synchronizing access control lists (ACLs) on platforms and filesystems that support them. In general, synchronization makes sense only in case both replicas support the same type of ACLs and recognize same users and groups. In some cases you may be able to go beyond that and synchronize ACLs to a replica that couldn't fully use them—this may be be useful for the purpose of preserving ACLs.

If one of the replicas does not support any type of ACLs then Unison will not attempt ACL synchronization. If the other replica does support ACLs then those will remain intact.

If both replicas support ACLs of any supported type then you can request Unison to try ACL synchronization (`acl` preference). Success of synchronization depends on permissions of the owner and group of Unison process (Unison must have permissions to set ACL) and the compatibility of ACL types on both replicas.

An ACL is propagated as a single unit, with all ACEs. There is no merging of ACEs from the replicas.

Caveat: ACE inheritance may in certain scenarios cause synchronization inconsistencies. In Windows, only explicit ACEs are synchronized; inherited ACEs are not actively synchronized, but Windows will propagate ACEs from parent directories (unless inheritance is explicitly prevented on a file or a directory—this prevention is also synchronized). Due to inheritance, the ultimately effective ACL may be different, or provide different access, even after synchronization.

Unison currently supports the following platforms and ACL types:

- Windows (Windows XP SP2 and later)
  - NTFS ACL (discrete ACL (DACL) only)
- Solaris, OpenSolaris and illumos-based OS (OpenIndiana, SmartOS, OmniOS, etc.)
  - NFSv4 ACL (ZFS ACL)
  - POSIX-draft ACL
  - Some NFSv4 ACL (ZFS ACL) cross-synchronization with POSIX-draft ACL

  - Full cross-synchronization with other platforms that support NFSv4 ACLs; limited cross-synchronization with POSIX-draft ACLs

- FreeBSD, NetBSD
  - NFSv4 ACL (ZFS ACL)
  - Limited POSIX-draft ACL (access ACL only; not default ACL)
  - Full cross-synchronization with other platforms that support NFSv4 ACLs

- Darwin (macOS)
  - Extended ACL

Not all filesystems on the listed platforms support all ACL types (or any ACLs at all).

Synchronizing POSIX ACLs on Linux is not supported directly. However, it is possible to synchronize these ACLs with another Linux system by synchronizing extended attributes (xattrs) instead, because POSIX ACLs are stored as xattrs by Linux. This is disabled by default (see Section 6.18 [Extended Attributes - xattrs]). A simple way to enable syncing POSIX ACLs on Linux is to enable the preference `xattrs` and add a preference `xattrignorenot` with a value `Path !system.posix_acl_*`. The `*` will be expanded to include both `posix_acl_access` and `posix_acl_default` attributes – if you only want to sync either one, just remove the `*` and type out the attribute name in full. If you want to prevent other xattrs from being synced then add an `xattrignore` with a value `Path *` (value `Regex .*` will also work).

6.18 Extended Attributes - xattrs

Unison allows synchronizing extended attributes on platforms and filesystems that support them. System attributes are not synchronized. What exactly is considered a system attribute is platform-dependent. Synchronization is possible cross-platform, but see caveats below.

If one of the replicas does not support extended attributes then Unison will not attempt attribute synchronization. If the other replica does support extended attributes then those will remain intact.

If both replicas support extended attributes then you can request Unison to try attribute synchronization (`xattrs` preference). Extended attributes from both replicas will not be merged, all extended attributes are propagated as a set from one replica to another.

Unison currently supports extended attributes on the following platforms:

- *Linux* Attributes in user, trusted and security namespaces. Synchronization of the latter two namespaces depends on `unison` process privileges and is disabled by default. To sync one or more attributes in the security namespace, for example, you can set the preference `xattrignorenot` to `Path !security.*` (for all) or to `Path !security.selinux` (for one specific attribute). Attributes in system namespace are not synchronized, with the exception of `system.posix_acl_default` and `system.posix_acl_access` (also disabled by default).

- *Solaris, OpenSolaris and illumos-based OS (OpenIndiana, SmartOS, OmniOS, etc.)*

- *FreeBSD, NetBSD* Attributes in user namespace.

- *Darwin (macOS)*

Not all filesystems on the listed platforms may support extended attributes.
Caveats:

- Some platforms and file systems support very large extended attribute values. Unison synchronizes only up to 16 MB of each attribute value.

- Attributes are synchronized as simple name-value pairs. More complex extended attribute concepts supported by some platforms are not synchronized.

- On Linux, attribute names always have a fully qualified form (`namespace.attribute`). Other platforms do not have the same constraint. The consequence of this is that Unison will sync the attribute names on Linux as follows: an `!` is prepended to the namespace name, except for the `user` namespace; the `user.` prefix is stripped from attribute names instead. This allows syncing extended attributes from Linux to other platforms. These transformations are reversed when syncing *to* Linux, resulting in correct fully qualified attribute names. The `xattrignore` and `xattrignorenot` preferences work on the transformed attribute names. This means that any patterns for the user namespace must be specified without the `user.` prefix and any patterns intended for other namespaces must begin with an `!`.

The `xattrignore` preference can be used to filter the names of extended attributes that will be synchronized. The most useful ignore patterns can be constructed with the `Path` form (where shell wildcards `*` and `?` are supported) and with the `Regex` form. The `xattrignorenot` preference can be used to override `xattrignore`.

Disabling the security and trusted namespaces on Linux is achieved by setting a default `xattrignore` pattern of `Regex !(security|trusted)[.].*`. Disabling the syncing of attributes used to store POSIX ACL on Linux is achieved by setting a default `xattrignore` pattern of `Path !system.posix_acl_*`.

6.19 Cross-Platform Synchronization

If you use Unison to synchronize files between Windows and Unix systems, there are a few special issues to be aware of.

Case conflicts. In Unix, filenames are case sensitive: `foo` and `FOO` can refer to different files. In Windows, on the other hand, filenames are not case sensitive: `foo` and `FOO` can only refer to the same file. This means that a Unix `foo` and `FOO` cannot be synchronized onto a Windows system — Windows won't allow two different files to have the "same" name. Unison detects this situation for you, and reports that it cannot synchronize the files.

You can deal with a case conflict in a couple of ways. If you need to have both files on the Windows system, your only choice is to rename one of the Unix files to avoid the case conflict, and re-synchronize. If you don't need the files on the Windows system, you can simply disregard Unison's warning message, and go ahead with the synchronization; Unison won't touch those files. If you don't want to see the warning on each synchronization, you can tell Unison to ignore the files (see Section 6.13 [Ignoring Paths]).

Illegal filenames. Unix allows some filenames that are illegal in Windows. For example, colons (':') are not allowed in Windows filenames, but they are legal in Unix filenames. This means that a Unix file `foo:bar` can't be synchronized to a Windows system. As with case conflicts, Unison detects this situation for you, and you have the same options: you can either rename the Unix file and re-synchronize, or you can ignore it.

6.20 Slow Links

Unison is built to run well even over relatively slow links such as modems and DSL connections.

Unison uses the "rsync protocol" designed by Andrew Tridgell and Paul Mackerras to greatly speed up transfers of large files in which only small changes have been made. More information about the rsync protocol can be found at the rsync web site (`http://samba.anu.edu.au/rsync/`).

If you are using Unison with `ssh`, you may get some speed improvement by enabling `ssh`'s compression feature. Do this by adding the option "`-sshargs -C`" to the command line or "`sshargs = -C`" to your profile.

6.21 Fast Update Detection

If your replicas are large and at least one of them is on a Windows system, you may find that Unison's default method for detecting changes (which involves scanning the full contents of every file on every sync—the only completely safe way to do it under Windows) is too slow. Unison provides a preference `fastcheck` that,

when set to `true`, causes it to use file creation times as 'pseudo inode numbers' when scanning replicas for updates, instead of reading the full contents of every file.

When `fastcheck` is set to `no`, Unison will perform slow checking—re-scanning the contents of each file on each synchronization—on all replicas. When `fastcheck` is set to `default` (which, naturally, is the default), Unison will use fast checks on Unix replicas and slow checks on Windows replicas.

This strategy may cause Unison to miss propagating an update if the modification time and length of the file are both unchanged by the update. However, Unison will never *overwrite* such an update with a change from the other replica, since it always does a safe check for updates just before propagating a change. Thus, it is reasonable to use this switch most of the time and occasionally run Unison once with `fastcheck` set to `no`, if you are worried that Unison may have overlooked an update.

Fastcheck is (always) automatically disabled for files with extension `.xls` or `.mpp`, to prevent Unison from being confused by the habits of certain programs (Excel, in particular) of updating files without changing their modification times.

6.22 Mount Points and Removable Media

Using Unison removable media such as USB drives can be dangerous unless you are careful. If you synchronize a directory that is stored on removable media when the media is not present, it will look to Unison as though the whole directory has been deleted, and it will proceed to delete the directory from the other replica—probably not what you want!

To prevent accidents, Unison provides a preference called `mountpoint`. Including a line like

```
mountpoint = foo
```

in your preference file will cause Unison to check, after it finishes detecting updates, that something actually exists at the path `foo` within both replicas; if it does not, the Unison run will abort.

Note that the preference's name is confusing; it is intended to be used when a root might or might not be mounted, but the value is a relative path within a root, for a file or directory that should be present. (The preference is not used to specify the path at which a replica is mounted.)