remsync, version 1.3

A remote synchronization utility
Edition 1.3, June 1994

by [No value for “Francois”] Pinard

Short Contents

1 Overview of `remsync` and friends 1
2 Specifications of program `remsync` 3
3 Specifications of other service programs 7
4 Related file formats 9
5 Various considerations 11

Table of Contents

1 Overview of `remsync` and friends 1

1.1 How `remsync` works 1
1.2 Quick start at using `remsync` 2

2 Specifications of program `remsync` 3

2.1 The `remsync` command and arguments 3
2.2 Automatic mechanisms in the `remsync` program 3
2.3 Commands for `remsync` 3

3 Specifications of other service programs 7

3.1 The `mailshar` command and arguments 7
3.2 The `mail-files` command and arguments 7
3.3 The `find-mailer` command and arguments 7

4 Related file formats 9

4.1 Format of the '`.remsync`' file 9
4.2 Format of transiting packages 10

5 Various considerations 11

5.1 Using News distribution instead? 11
5.2 Documentation for obsolete scripts 11
5.2.1 mailsync 11
5.2.2 resync 12

1 Overview of remsync and friends

The remsync program allows for transmitting, over email, selected parts of directories for trying to maintain up-to-date files over many sites. It sends out and processes incoming specially packaged files using shar, tar, gzip and electronic mail programs.

There is no *master* site, each site has an equal opportunity to modify files, and modified files are propagated. Among many other commands, the broadcast command sends an update package from the current site to all others, the process command is used to apply update packages locally after reception from remote sites.

The unit of transmission is whole files. For now, whenever a module is modified, it is silently synchronized only if it has been modified at only one place. The merging has to be done at the site where the discrepancy is observed, from where it is propagated again.

1.1 How remsync works

How does remsync keep track of what is in sync, and what isn't? See Section 4.1 [Xremsync], page 9, for a the documentation on the '.remsync' file format. I understand that a mere description of the format does not replace an explanation, but in the meantime, you might guess from the format how the program works.

All files are summarized by a checksum, computed by the sum program. There are a few variants of sum computing checksums in incompatible ways, under the control of options. remsync attempts to retrieve on each site a compatible way to do it, and complains if it cannot.

remsync does not compare dates or sizes. Experience shown that the best version of a file is not necessarily the one with the latest timestamp. The best version for a site is the current version on this site, as decided by its maintainer there, and this is this version that will be propagated.

Each site has an idea of the checksum of a file for all other sites. These checksums are not necessarily identical, for sites do not necessarily propagate to all others, and the propagation network maybe incomplete or asymmetrical in various ways.

Propagation is never done unattended. The user on a site has to call remsync broadcast to issue synchronization packages for other sites. If this is never done, the local modifications will never leave the site. The user also has to call remsync process to apply received synchronization packages. Applying a package does not automatically broadcast it further (maybe this could change?).

If a site A propagates some files to sites B and D, but not C, site B is informed that site D also received these files, and site D is informed that site B also received these files, so they will not propagate again the same files to one another. However, both site B and D are susceptible to propagate further the same files to site C.

It may happen that a site refuses to update a file, or modifies a file after having been received, or merges versions, or whatever. So, sites may have a wrong opinion of the file contents on other sites. These differences level down after a few exchanges, and it is very unlikely that a file would not be propagated when it should have.

This scheme works only when the various people handling the various files have confidence in one each other. If site B modifies a file after having received it from site A, the file will eventually be propagated back to site A. If the original file stayed undisturbed on site A, that is, if remsync proves that site B correctly knew the checksum of the original file, then the file will be replaced on site A without any user confirmation. So, the user on site A has to trust the changes made by the user on site B.

If the original file on site A had been modified after having been sent in a synchronization package, than it is the responsibility of the user on site A to correctly merge the local modifi-

cations with the modifications observed in the file as received from site *B*. This responsibility is real, since the merged file will later be propagated to the other sites in an authoritative way.

1.2 Quick start at using remsync

2 Specifications of program remsync

2.1 The remsync command and arguments

At the shell prompt, calling the command `remsync` without any parameters initiates an interactive dialog, in which the user types commands and receives feedback from the program.

The command `remsync`, given at the shell prompt, may have arguments, in which case these arguments taken together form one `remsync` interactive command. However, '`--help`' and '`--version`' options are interpreted especially, with their usual effect in GNU. Once this command has been executed, no more commands are taken from the user and `remsync` terminates execution. This allows for using `remsync` in some kind of batch mode. It is unwise to redirect `remsync` standard input, because user interactions might often be needed in ways difficult to predict in advance.

The two most common usages of `remsync` are the commands:

```
remsync b
remsync p
```

The first example executes the `broadcast` command, which sends synchronization packages to all connected remote sites for the current local directory tree.

The second example executes the `process` command, which studies and complies with a synchronisation package saved in the current directory (not necessarily into the synchronized directory tree), under the usual file name '`remsync.tar.gz`'.

2.2 Automatic mechanisms in the remsync program

The following points apply to many of the `remsync` commands. We describe them here once and for all.

- The file '`.remsync`' describes the various properties for the current synchronization. It is kept right in the top directory of a synchronized directory tree. Some commands may be executed without any need for this file. The program waits as far as possible before reading it.
- If the '`.remsync`' file is not found when required, and only then, the user is interactively asked to fill a questionnaire about it.
- If the '`.remsync`' file has been logically modified after having been read, or if it just has been created, the program will save it back on disk. But it will do so only before reading another '`.remsync`' file, or just before exit. A preexisting '`.remsync`' will be renamed to '`.remsync.bak`' before it is rewritten, when this is done, any previous '`.remsync.bak`' file is discarded.
- Many commands refer to previously entered information by repeating this information. For example, one can refer to a particular `scan` statement by entering the wildcard to be scanned by this statement. An alternative method of specifying a statement consists in using the decimal number which appears between square brackets in the result of a `list` command.
- Whenever a site list must be given, it is a space separated list of remote sites. If the list is preceeded by a bang (!), the list is complemented, that is, the sites that will be operated upon are all those *not* appearing in the list. As a special case, if the site list is completely empty, then all sites are selected.

2.3 Commands for remsync

Program commands to `remsync` may be given interactively by the user sitten at a terminal. They can come from the arguments of the `remsync` call at the shell level. Internally, the `process` command might obey many sub-commands found in a received synchronization package.

Program commands are given one per line. Lines beginning with a sharp (#) and white lines are ignored, they are meant to increase clarity or to introduce user comments. With only a few exceptions, commands are introduced by a keyword and often contains other keywords. In all cases, the keywords specific to `remsync` may be abbreviated to their first letter. When there are many keywords in succession, the space separating them may be omitted. So the following commands are all equivalent:

```
list remote
l remote
list r
l r
listremote
lr
```

while the following are not legal:

```
l rem
lisremote
```

Below, for clarity, keywords are written in full and separated by spaces. Commands often accept parameters, which are then separated by spaces. All available commands are given in the table. The first few commands do not pre-require the file '`.remsync`'. The last three commands are almost never used interactively, but rather automatically triggered while `process`'ing received synchronization packages.

`?`

Display a quick help summary of available commands.

`!` [*shell-command*]

If *shell-command* has been given, execute it right now as a shell command. When not given, rather start an interactive shell. Exiting from the shell will return to this program. The started shell is taken from the `SHELL` environment variable if set, else `sh` is used.

`quit`

Leave the program normally and return to the shell.

`abort`

Leave the program with a nonzero exit status and return to the shell. No attempt is made to save a logically modified '`.remsync`' file.

`visit` *directory*

Select another synchronized directory tree for any subsequent operation. *directory* is the top directory of the synchronized directory tree.

`process` [*file*]
`list` [*type*]

List all known statements about some information *type*. Allowable keywords for *type* are `local`, `remote`, `scan`, `ignore` and `files`. The keyword `files` asks for all empty statements (see later). If *type* is omitted, then list all known statements for all types, except those given by `files`.

[`create`] *type value*

Create a new statement introducing a *value* for a given *type*. Allowable keywords for *type* are `remote`, `scan` and `ignore`. The `create` keyword may be omitted.

For `create ignore`, when the pattern is preceeded by a bang (!), the condition is reversed. That is, only those files which do match the pattern will be kept for synchronization.

`delete` *type value*
: Delete an existing statement supporting some *value* for a given *type*. Allowable keywords for *type* are `remote`, `scan` and `ignore`.

`email` *remote value*
: Modify the electronic mail address associated with some *remote* site, giving it a new *value*. The special `local` keyword for *remote* may be used to modify the local electronic mail address.

`home` *remote value*
: Modify the top directory of the synchronized directory tree associated with some *remote* site, giving it a new *value*. The special `local` keyword for *remote* may be used to modify the local top directory.

`broadcast` *site_list*
: Send by electronic mail an update package to all sites from *site_list*, containing for each site all and only those files which are known to be different between the remote site and here.

`version` *version*
: This command is not meant for interactive use. It establishes the `remsync` version needed to process the incoming commands.

`from` *site_list*
: This command is not really meant for interactive use. The first site from the *site_list* is the remote site which originated the synchronization package. All the others are all the sites, including here, which were meant to be synchronized by the `broadcast` command that was issued at the originating remote site.

`sum` *file checksum*
: This command is not really meant for interactive use. It declares the *checksum* value of a particular *file* at the originating remote site. Also, if at least one `sum` command is received, then it is guaranteed that the originating remote site sent one `sum` command for each and every file to be synchronized, so any found local file which was not subject of any `sum` command does not exist remotely.

`if` *file checksum packaged*
: This command is not really meant for interactive use. It directs the `remsync` program to check if a local *file* has a given *checksum*. If the checksum agrees, then the local file will be replaced by the *packaged* file, as found in the received synchronization invoice.

3 Specifications of other service programs

3.1 The `mailshar` command and arguments

3.2 The `mail-files` command and arguments

3.3 The `find-mailer` command and arguments

4 Related file formats

4.1 Format of the '.remsync' file

The '.remsync' file saves all the information a site needs for properly synchronizing a directory tree with remote sites. Even if it is meant to be editable using any ASCII editor, it has a very precise format and one should be very careful while modifying it. The '.remsync' file is better handled through the `remsync` program and commands.

The '.remsync' file is made up of statements, one per line. Each line begins with a statement keyword followed by a single ⟨TAB⟩, then by one or more parameters. The keyword may be omitted, in this case, the keyword is said to be *empty*, and the line begins immediately with the ⟨TAB⟩. After the ⟨TAB⟩, if there are two parameters or more, they should all be separated with a single space. There should not be any space between the last parameter and the end of line (unless there are explicit empty parameters).

The following table gives the possible keywords. Their order of presentation in the table is also the order of appearance in the '.remsync' file.

`remsync` This statement identifies the '.remsync' format. The only parameter states the file format version.

`local` This statement should appear exactly once, and has exactly two parameters. The first parameter gives the electronic mail address the other sites should use for sending synchronization packages here. The second parameter gives the name of the local directory tree to synchronize, in absolute notation.

`remote` This statement may appear zero, one or more times. Each occurrence connects the synchronized directory tree to another tree on a remote site. The first parameter gives one electronic mail address where to send remote synchronization packages. The second parameter gives the name of the corresponding directory tree for this remote electronic mail address, in absolute notation.

`scan` This statement may appear zero, one or more times. When it does not appear at all, the whole local directory tree will always be scanned, searching for files to synchronize. When the statement appears at least once, the whole local directory tree will not be scanned, but only those files or directories appearing in one of these statements. Each `scan` statement has exactly one parameter, giving one file or directory to be studied. These are usually given relative to top directory of the local synchronization directory tree. Shell wildcards are acceptable.

`ignore` This statement may appear zero, one or more times. Each occurrence has one parameter giving a regular expression, according to Perl syntax for regular expressions. These *regexp*s are applied against each file resulting from the scan. If any of the `ignore` expression matches one of resulting file, the file is discarded and is not subject to remote synchronization.

After all the statements beginning by the previous keywords, the '.remsync' file usually contains many statements having the empty keyword. The empty keyword statement may appear zero, one or more times. Each occurrence list one file being remotely synchronized. The first parameter gives an explicit file name, usually given relative to the top directory of the local synchronized directory tree. Shell wildcards are *not* acceptable.

Besides the file name parameter, there are supplementary parameters to each empty keyword statement, each corresponding to one remote statement in the '.remsync' file. The second parameter corresponds to the first remote, the third parameter corresponds to the second remote,

etc. If there are more remote statements than supplementary parameters, missing parameters are considered to be empty.

Each supplementary parameter usually gives the last known checksum value for this particular file, as computed on its corresponding *remote* site. The parameter contains a dash `-` while the remote checksum is unknown. The checksum value for the *local* copy of the file is never kept anywhere in the '`.remsync`' file. The special value '`666`' indicates a checksum from hell, used when the remote file is known to exist, but for which contradictory information has been received from various sources.

4.2 Format of transiting packages

5 Various considerations

5.1 Using News distribution instead?

One correspondent thinks that perhaps the news distribution mechanism could be pressed into service for this job. I could have started from C-news, say, instead of from scratch, and have progressively bent C-news to behave like I wanted.

My feeling is that the route was shorter as I did it, from scratch, that it would have been from C-news. Of course, I could have removed the heavy administrative details of C-news: the history and `expire`, the daemons, the `cron` entries, etc., then added the interactive features and specialized behaviors, but all this clean up would certainly have took energies. Right now, non counting the subsidiary scripts and shar/unshar sources, the heart of the result is a single (1200 lines) script written in Perl, which I find fairly more smaller and maintainable than a patched C-news distribution would have been.

5.2 Documentation for obsolete scripts

This is merely a place holder for previous documentation, waiting that I clean it up. You have no interest in reading further down.

5.2.1 mailsync

```
Usage: mailsync [ OPTION ] ... [ EMAIL_ADDRESS ] [ DIRECTORY ]
   or: mailsync [ OPTION ] ... SYNC_DIRECTORY
```

Option -i simply sends a `ihave` package, with no bulk files. Option -n inhibits any destructive operation and mailing.

In the first form of the call, find a synchronisation directory in DIRECTORY aimed towards some EMAIL_ADDRESS, then proceed with this synchronisation directory. EMAIL_ADDRESS may be the name of a file containing a distribution list. If EMAIL_ADDRESS is not specified, all the synchronisation directories at the top level in DIRECTORY are processed in turn. If DIRECTORY is not specified, the current directory is used.

In the second form of the call, proceed only with the given synchronisation directory SYNC_DIRECTORY.

For proceeding with a synchronisation directory, whatever the form of the call was, this script reads the `ident` files it contains to set the local user and directory and the remote user and directory. Then, selected files under the local directory which are modified in regard to the corresponding files in the remote directory are turned into a synchronisation package which is mailed to the remote user.

The list of selected files or directories to synchronize from the local directory are given in the `list` file in the synchronisation directory. If this `list` file is missing, all files under the local directory are synchronized.

What I usually do is to `cd` at the top of the directory tree to be synchronized, then to type `mailsync` without parameters. This will automatically prepare as many synchronisation packages as there are mirror systems, then email multipart shars to each of them. Note that the synchronisation package is not identical for each mirror system, because they do not usually have the same state of synchronisation.

`mailsync` will refuse to work if anything needs to be hand cleaned from a previous execution of `mailsync` or `resync`. Check for some remaining '`_syncbulk`' or '`_synctemp`' directory, or for a '`_syncrm`' script.

```
TODO:
- interrogate the user if 'ident' file missing.
```

```
- automatically construct the local user address.
- create the synchronisation directory on the fly.
- avoid duplicating work as far as possible for multiple sends.
- have a quicker mode, depending on stamps, not on checksums.
- never send core, executables, backups, '.nsf*', '*/_synctemp/*', etc.
```

5.2.2 resync

```
Usage: resync [ OPTION ]... TAR_FILE
   or: resync [ OPTION ]... UNTARED_DIRECTORY
```

Given a tar file produced by mailsync at some remote end and already reconstructed on this end using unshar, or a directory containing the already untared invoice, apply the synchronization package locally.

Option -n inhibits destroying or creating files, but does everything else. It will in particular create a synchronization directory if necessary, produce the '`_syncbulk`' directory and the '`_syncrm`' script.

The synchronization directory for the package is automatically retrieved or, if not found, created and initialized. `resync` keeps telling you what it is doing.

There are a few cases when a resync should not complete without manual intervention. The common case is that several sites update the very same files differently since they were last resync'ed, and then mailsync to each other. The prerequisite checksum will then fail, and the files are then kept into the '`_syncbulk`' tree, which has a shape similar to the directory tree in which the files where supposed to go. For GNU Emacs users, a very handy package, called emerge, written by Dale Worley <drw@kutta.mit.edu>, helps reconciling two files interactiveley. The '`_syncbulk`' tree should be explicitely deleted after the hand synchronisation.

Another case of human intervention is when files are deleted at the mailsync'ing site. By choice, all deletions on the receiving side are accumulated in a '`_syncrm`' script, which is not executed automatically. Explicitely executed, '`_syncrm`' will remove any file in the receiving tree which does not exist anymore on the sender system. I often edit '`_syncrm`' before executing it, to remove the unwanted deletions (beware the double negation :-). The script removes itself.

All the temporary files, while resynchronizing, are held in '`_synctemp`', which is deleted afterwards; if something goes wrong, this directory should also be cleaned out by hand. `resync` will refuse to work if anything remains to be hand cleaned.

TODO:
- interrogates the user if missing receiving directory in '`ident`'.
- allow '`remote.sum`' to be empty or non-existent.