librsync programmer's manual 1.0pre DRAFT

Martin Pool

20th July 2004

Contents

1 Introduction

The librsync library implements network delta-compression of streams and files.

librsync was originally used in the rproxy experiment in delta-compression for HTTP. One popular application is rdiff-backup, which uses rdiff deltas for incremental backups.

`rdiff` is a command-line scriptable interface to librsync. The rdiff manual explains the concepts of network deltas and should be read first.

librsync can be downloaded from http://librsync.sourcefrog.net/ and used, modified and redistributed under the terms of the GNU Lesser General Public License (version 2.1 or later). That licence also applies to this manual.

The library provides four basic operations:

gensig Generate a signature describing a file, from which deltas may later be generated.

loadsig Load a signature into memory.

delta Calculate a delta from an in-memory signature to a new file, and write the resulting delta to a file.

patch Read a delta from a file and apply it to a basis file, producing an output file.

These correspond to the basic operations of `rdiff`, except that loading a signature is done as a separate operation from calculating a delta.

librsync provides a high-level interface for applications that just want to make and use signatures and deltas with a single function call.

Alternatively there is a streaming interface which can support blocking or non-blocking IO and processing of encapsulated, encrypted or compressed streams. Each operation in progress is represented by an `rs_job_t` opaque structure, called a *job object*. Any number of jobs can be in progress at the same time. Through this interface the application creates a job object, supplies IO callbacks, and then repeated calls into librsync to run the job.

2 Interface conventions

The public interface to librsync is defined in `librsync1.h`. All external symbols have the prefix `rs_`, or `RS_` in the case of preprocessor symbols.

Symbols beginning with `rs__` (double underscore) are private and should not be called from outside the library.

3 Return codes

Almost all functions in librsync return a value of the enumerated type `rs_result`. The values include:

`RS_DONE` == **0** Processing is complete.

`RS_BLOCKED` The operation cannot proceed at the moment.

`RS_RUNNING` The job is still running. (This should never be returned to the application.)

`RS_IO_ERROR` Input/output error.

`RS_SYNTAX_ERROR` Command-line syntax error.

`RS_MEM_ERROR` Out of memory or similar conditions.

`RS_INPUT_ENDED` Input ended abruptly, typically because of a truncated file or dropped network connection.

`RS_BAD_MAGIC` Wrong magic number at the start of an input file. The input file is probably not in the right format, or perhaps is from an incompatible version of librsync.

`RS_UNIMPLEMENTED` The author is lazy.

`RS_CORRUPT` Unbelievable value in input stream.

`RS_INTERNAL_ERROR` Probably a library bug, or perhaps invalid input.

`RS_PARAM_ERROR` Bad value passed to library. Probably an application bug.

`RS_EOF` Reading from a file reached end-of-file.

There are a few cases where bugs in the library may cause it to abort the process. These should never occur once bugs have been eliminated from the application and library.

Given an `rs_result` value, `rs_strerror` returns a read-only human-readable string describing the error:

```
char const *rs_strerror(rs_result result);
```

(The current code always returns an English string, but it should probably be returned in the message current locale.)

4 Processing whole files

librsync provides a high-level API for processing whole files. These functions open files, process the entire contents, and return an overall result.

Some applications do not require fine-grained control over IO, but rather just want to process a whole file with a single call. librsync provides "whole-file" functionality to do exactly that. The whole-file operations are the core of the `rdiff` program.

`rdiff_loadsig_files` generates a signature of `input_file` and writes it to `output_file`. The signature is generated using the given block and strong sum:

```
rs_result
rs_loadsig_files(const char *sig_file,
                 rs_signature_t **sig_out);

rs_result
rs_files_sig(const char *input_file,
             const char *output_file,
```

```
                size_t block_len,
                size_t strong_len,
                rs_stats_t *stats)

rs_result
rs_delta_files(rs_signature_t *sig,
                  const char *input_file,
                  const char *output_file);
```

The signature is generated using the given block and strong sum lengths. Default values are used if zero is given for these two parameters.

If `stats` is not null, statistics are returned in the given statistics structure.

5 Debug messages

librsync can optionally produce a error/debug trace while it runs. Error messages supplement return codes by describing in more detail what went wrong. Debug messages are useful when debugging librsync or applications that call it.

The default configuration is that warning and error messages are written to stderr. This should be appropriate for many applications. If it is not, the level and destination of messages may be changed.

Messages are passed out of librsync through a trace callback which is passed a severity and message string. The type for this callback is:

```
typedef void rs_trace_fn_t(int level, char const *msg);
```

The default trace function is:

```
void rs_trace_stderr(int level, char const *msg);
```

The trace callback may be changed at runtime:

```
void rs_trace_to(rs_trace_fn_t *trace_fn);
```

Messages from librsync are labelled with a severity indicator of enumerated type `rs_loglevel`:

`RS_LOG_CRIT` Critical error such as hitting an unimplemented case in librsync. librsync may abort the process if it cannot return safely.

`RS_LOG_ERR` Serious error. The current operation has probably failed.

`RS_LOG_WARNING` A problem was encountered but it has not interrupted processing.

`RS_LOG_INFO` Information on normal progress. May be suitable for a `--verbose` mode.

`RS_LOG_DEBUG` Very detailed internal debug information. Useful when debugging librsync or programs that call it.

The application may also specify a minimum severity of interest. The default level is `RS_LOG_INFO`. Messages lower than the specified level are discarded without being passed to the trace callback:

```
void rs_trace_set_level(rs_loglevel level);
```

6 IO callbacks

librsync jobs use IO callbacks to read and write files. These callbacks might write the data directly to a file or network connection, or they might do some additional work such as compression or encryption.

Callbacks are passed a *baton*, which is chosen by the application when setting up the job. The baton can hold context or state for the callback, such as a file handle or descriptor.

There are three types of callbacks, for input, output, and a special one for random-access reads of the basis file when patching. Different types of job use different callbacks. The callbacks are assigned when the job is created and cannot be changed. (If the behaviour of the callback needs to change during the job, that can be controlled by variables in the baton.)

There are three function typedefs for these callbacks:

```
typedef rs_result rs_cb_read(void *baton,
                             char *buf,
                             size_t buf_len,
                             size_t *bytes_read);

typedef rs_result rs_cb_basis(void *baton,
                              char *buf,
                              size_t buf_len,
                              off_t offset,
                              size_t *bytes_read);

typedef rs_result rs_cb_write(void *baton,
                              const char *buf,
                              size_t buf_len,
                              size_t *bytes_written);
```

IO callbacks are passed the address of a buffer allocated by librsync which they read data into or write data from, plus the length of the buffer.

Callbacks return an `rs_result` value to indicate success, an error, or being blocked. Callbacks must set the appropriate `bytes_read` or `bytes_written` to indicate how much data was processed. They may process only part of the requested data, in which case they still return `RS_DONE`. In this case librsync will call the callback again later until it either completes, fails, or blocks.

When a read callback reaches end-of-file and can return no more data, it should return `RS_EOF`. In this case no data should be returned; the output value of bytes_read is ignored. If the callback has just a little data left before end of file, then it should return that data with `RS_DONE`. On the next call, unless the file has grown, it can return `RS_EOF`.

If the callbacks return an error, that error will typically be passed back to the application.

IO callbacks are only called from within `rs_job_run`, never spontaneously. Different callbacks may be called several times in a single invocation of `rs_job_run`.

6.1 stdio callbacks

librsync provides predefined IO callbacks that wrap the C stdio facility. The baton argument for all these functions is a `FILE*`:

```
rs_result rs_cb_read_stdio(void*,
                           char *buf,
                           size_t buf_len,
                           size_t *bytes_read);

rs_result rs_cb_basis_stdio(void *,
                            char *buf,
                            size_t buf_len,
                            off_t offset,
                            size_t *bytes_read);

rs_result rs_cb_write_stdio(void *voidp,
                            const char *buf,
                            size_t buf_len,
                            size_t *bytes_written);
```

There is also a utility function that wraps `fopen`. It reports any errors through the librsync error log, and translates return values. It also treats `-` as stdin or stdout as appropriate.

```
rs_result rs_stdio_open(const char *file,
                        const char *mode,
                        FILE **filp_out);
```

7 Creating Jobs

Jobs are created by calling `rs_gensig_begin`, `rs_delta_begin`, `rs_loadsig_begin` or `rs_patch_begin`. These functions create a new job object, which can then be run using `rs_job_run`.

```
rs_result rs_gensig_begin(rs_job_t **job_out,
                          size_t block_len,
                          size_t strong_sum_len,
                          rs_cb_read *read_cb, void *read_baton,
                          rs_cb_write *write_cb, void *write_baton);
```

A newly allocated job object is stored in `*job_out`.

The patch job accepts the patch as input, and uses a callback to look up blocks within the basis file.

You must configure read, write and basis callbacks after creating the job but before it is run.

After creating the job, call `rs_job_run` to feed in patch data and retrieve output data. When the job is complete, call `rs_job_finish` to dispose of the job object and free memory.

8 Running Jobs

The work of the operation is done when the application calls `rs_job_run`. This includes reading from input files via the callback, running the rsync algorithms, and writing output.

The IO callbacks are only called from inside `rs_job_run`. If any of them return an error, `rs_job_run` will generally return the same error.

When librsync needs to do input or output, it calls one of the callback functions. `rs_job_run` returns when the operation has completed or failed, or when one of the IO callbacks has blocked.

`rs_job_run` will usually be called in a loop, perhaps alternating librsync processing with other application functions.

```
rs_result rs_job_run(rs_job_t *job);
```

9 Deleting Jobs

A job is deleted and its memory freed up using `rs_job_free`:

```
rs_result rs_job_free(rs_job_t *job);
```

This is typically called when the job has completed or failed. It can be called earlier if the application decides it wants to cancell processing.

`rs_job_free` does not delete the output of the job, such as the sumset loaded into memory. It does delete the job's statistics.

10 Non-blocking IO

The librsync interface allows non-blocking streaming processing of data. This means that the library will accept input and produce output when it suits the application: the library should not ever block waiting for IO. It is intended to be usable in threaded programs, but does not require threading. librsync should work with programs that do either blocking or nonblocking IO.

Normally callbacks will read/write the whole buffer when they're called, but in some cases they might not be able to process all of it, or perhaps not process any at all. This might happen if the callbacks are connected to a nonblocking socket. Either of two things can happen in this case. If the callback returns `RS_BLOCKED`, then `rs_job_run` will also return `RS_BLOCKED` shortly.

When an IO callback blocks, it is the responsibility of the application to work out when it will be able to make progress and therefore when it is worth calling `rs_job_run` again. Typically this involves a mechanism like `poll` or `select` to wait for the file descriptor to be ready.

11 Job Statistics

Jobs accumulate statistics while they run, such as the number of input and output bytes. The particular statistics collected depend on the type of job.

```
const rs_stats_t * rs_job_statistics(rs_job_t *job);
```

`rs_job_statistics` returns a pointer to statistics for the job. The pointer is valid throughout the life of the job, until the job is freed. The statistics are updated during processing and can be used to measure progress.

Statistics can be written to the trace file in human-readable form:

```
int rs_log_stats(rs_stats_t const *stats);
```

Statistics are held in a structure referenced by the job object. The statistics are kept up-to-date as the job runs and so can be used for progress indicators.

12 Utility functions

Some additional functions are used internally and also exposed in the API:

- encoding/decoding binary data: `rs_base64`, `rs_unbase64`, `rs_hexify`.
- MD4 message digests: `rs_mdfour`, `rs_mdfour_begin`, `rs_mdfour_update`, `rs_mdfour_result`.