librascal: a connection library

Justin Forest <justin.forest@gmail.com>

Table of Contents

librascal: a connection library....1
- Abstract....3
- Introduction....3
- Implementation....3
  - Request processing....3
  - Event processing....3
  - Error handling....4
  - Data flow buffering....4
  - Memory management....4
  - Domain name and service resolution functions....4
  - Threading model....5
- Pending changes....5
- Type Reference....5
  - addr_t....5
  - sock_t....6
  - rrid_t....6
- Function reference....6
  - rascal_accept....6
  - rascal_accept_service....6
  - rascal_aton....7
  - rascal_cancel....7
  - rascal_connect....7
  - rascal_connect_service....8
  - rascal_dispatcher....9
  - rascal_getaddr....9
  - rascal_gethost....10
  - rascal_get_rq_size....10
  - rascal_get_sq_size....10
  - rascal_get_errmsg....10
  - rascal_init....10
  - rascal_isok....11
  - rascal_ntoa....11
  - rascal_read....11
  - rascal_reads....12
  - rascal_set_context....12
  - rascal_wait....12
  - rascal_work....13
  - rascal_write....13
- Advanced functions....13
  - rascal_get_option....13
  - rascal_set_dispatcher....13
  - rascal_set_nameserver....14

rascal_set_option....14
rascal_shrink....14
Internal reference....15
datachain....15
pageman....15
stock....16

Abstract

librascal is a library designed to help building high-performance networking applications. This is the technical reference for the library.

Introduction

librascal stands for **r**ealtime **as**ynchronous **c**onnection **a**bstraction **l**ayer. It is a library that provides functions for building portable, high-performance networking applications. The following must be taken into consideration when deciding to start using the library:

1. The library is fully asynchronous and performs no blocking operations, which makes it possible to use in time critical environments. All requests either complete immediately or a completion notification is delivired using events
2. The library supports multithreading.
3. The library performs unconditional data flow buffering in both directions.
4. The library performs caching of all internally allocated objects to decrease the overhead of the memory allocation process.
5. The library provides functions for working with DNS, and doing so in a simple and effective manner. The library makes use of SRV resource records to help achieve best user experience.
6. The library supports IPv6 on systems which have the sixth stack installed.
7. The library is open source and is licensed under the terms of GNU LGPL.

Implementation

Functions and data types required to use the library are declared in header file rascal.h. The library is written in C++; however, considering the "pure C" API, it can be used by C programs just as well. The library is only available as a dynamic library, static linking is not supported.

The naming convention used in this documentation uses the term "application" to refer to a program that is using functions provided by the library.

Request processing

The library is completely asynchronous. This means that a call to a library function returns immediately, always. Operations that take time to complete are scheduled for background execution and return a handle to the operation. The `rascal_isok` function must first be used on these handles to make sure that the operation has been successfully scheduled. Any of the following actions can then be performed:

1. Cancel the operation by calling the `rascal_cancel` function.
2. Store the handle for later use.
3. Ignore the handle, if no interaction with the request is planned.

Event processing

Because the library is realtime, no function may ever block the execution. Therefore all functions return immediately, either with an immediate result available, or scheduling the operation for background execution.

Events are delivered to a user defined callback function called event dispatcher. The connection that the event is delivered for is blocked for write operations during the time of dispatcher's execution. This means that all requests to write to the connection are scheduled, but the actual writing will start as soon as the event dispatcher finishes executing and returns control to the library. (This is done to optimize network performance when large amounts of small data portions are written.)

Error handling

The library does not use neither exceptions nor anything else but request identifiers to indicate errors. What this (also) means is that the library does not handle neither exceptions nor signals nor anything else. If you pass a bad pointer to a function and its execution is interrupted, further execution might be impossible due to internal objects left locked.

Human readable error messages corresponding to error codes returned by the library can be retreived using the `rascal_get_errmsg` function.

Data flow buffering

All data transfer functions make use of the internal per-connection buffer to optimize performance and prevent the application from implementing its own complicated memory management. The incoming data flow is accumulated in the incoming buffer and can be partially or fully reterived using functions `rascal_read` and `rascal_reads`. The `rascal_reads` function was designed to simplify working with text based protocols, such as SMTP or IRC: it retrieves a single line of text from the head of the FIFO buffer.

The output function, `rascal_write`, appends the data to the output buffer of a connection.

Note that all transfer functions copy the data either to or from the internal buffer. There are no functions to receive data directly into user defined area of memory or to send from there.

Note also that the data is sent and received by portions, each several kilobytes in size. Keep this in mind when reading large amounts of data of a known size, for which you can allocate enough memory. Also keep that in mind when writing large amounts of data that you anyway keep in memory. The size of the internal cache is not limited, but frequent expansion can lead to poor performance and inoptimal resource usage; it is preferred that the application sends each portion of data as soon as a `rop_write` event is reported (see below).

Memory management

The library performs caching of internally allocated objects. This helps reducing the overhead of memory allocation (which is a complex procedure and involves things like thread locking and defragmentation). The size of the cache pool can increase on the beginning of the application's networking activity, but it stabilizes and stops growing as the network load does. The library does not provide the application with a way to determine the size of the pool; assume that the memory management is optimal. The library does, however, provide a function to empty the pool of cached objects, which should only be used after burst operations which happen rarely. Cleaning the pool too often leads to poor performance.

The library uses standard `malloc` and `free` functions, and there is no way the application can specify its own memory allocation functions. This is made to guarantee consistent results of memory allocation by using a trusted source.

Domain name and service resolution functions

Though the main purpose of the library is working with data oriented TCP/IP connections, domain name resolution is another vital part because it is seldom the case when the numeric address of the destination connection endpoint is known to the user. The library provides a limited set of functions sufficient to resolve hostnames and locate services using DNS. The preferred way to establish a connection is to use *service discovery*, a technology that provides failure safety, load balancing and other ways to improve the user experience (see RFC2782 for details). The library provides a function for locating a service and connecting to it, `rascal_connect_service`, and a function to install a local service handler, `rascal_accept_service`.

For traditional DNS lookup there are functions `rascal_gethost` and `rascal_getaddr`. The library relies on a recursive name server (the vast majority of end-user name servers, such as ones operated by ISPs, are recursive).

The library supports multiple name servers for the purpose of failure safety. The list of available servers is read from system settings when the library is initialized. If an application needs to use a different set of name servers, it can be done using the `rascal_set_nameserver` function.

Threading model

The library is threads safe. There are three modes of execution, guaranteed to satisfy the most developers' needs: non-threaded, single threaded and thread-per-CPU. For threaded modes of execution, the application might need to use a thread locking mechanism, depending on the data isolation model being used by the application. The library does not provide such functions; however, file `common/mutex.h` contains an example of a portable implementation of thread locking, which is used internally by the library and is guaranteed to work on all supported systems.

Pending changes

This chapter describes the implementation and design details that need to be reviewed and/or changed. Discussion on this topic is welcome at e-mail.

- *Reading into the specified memory area*, to avoid copying large amounts of data. Transferring big binary objects will benefit from this.
- *Writing from the specified memory area* would also be needed, obviously.
- *Design recommendations* might be a good section for the documentation. Namely, things like chosing a threading model would be described there. Also, it would note whether MFC, VCL and other popular libraries go into alertable state when saving time while waiting for an event to occur; if they do, it would mean that the whole asynchronous application can be built within just one thread.
- A research on the overhead of frequently calling ReadFileEx frequently with small data blocks versus calling it less frequently with larger data blocks. Nihil would mean that `rascal_read` would just call `rascal_read_ex` and maintain an internal buffer.
- Declare functions for transparent data flow compression.
- The return value of a successfull call to `rascal_read` and related functions should be *zero*. Passing zero to `rascal_cancel` must result in an "invalid handle" error.

Type Reference

This section documents all data types and structures used by the library. For C programs, each structure has a corresponding typedef with the same name.

addr_t

This is an opaque structure used to describe an internet address.

```
struct addr_t {
        unsigned int length;
        unsighed char data[16];
};
```

The application has no way of learning the type of the address other than guessing it by the value of the `length` member. Typically, the value is 4 for an IPv4 address and 16 for an IPv6 address. Other address families are currently not supported.

To convert an address to a human-readable string, the application can use the `rascal_ntoa` function; to convert a string to an `addr_t` structure, the `rascal_aton` function should be used. Otherwise, the application can work with address descriptors on the octet level, by accessing the `data` array.

The following additional methods are available in C++ mode:

```
// Constructors:
addr_t(unsigned int length = 0);
addr_t(const addr_t &src);
// Operators:
addr_t& operator = (const addr_t &src);
```

sock_t

This structure extends the opaque address descriptor `addr_t` with a port number:

```
struct sock_t {
        struct addr_t addr;
        unsigned short port;
};
```

The following additional methods are available in C++ mode:

```
// Constructors:
sock_t(unsigned short _port = 0);
sock_t(const sock_t &src);
sock_t(addr_t &src, unsigned short _port);
// Operators:
operator sock_t* ();
```

rrid_t

This type is used as a request identifier. It also serves as a success indicator; the value can be tested using the `rascal_isok` function to see if the requested operation was successfully scheduled or performed. If the result is negative, then the corresponding error description can be retrieved using the `rascal_get_errmsg` function.

Request identifiers can be used to cancel a request that was scheduled for background execution. However, not every operation can be cancelled; an example of one that can't be is `rascal_write`. Typically, a manifest constant `REC_SUCCESS` is returned when the request was successfully scheduled but can not be cancelled.

Function reference

This sections is a complete function reference. The calling convention for all functions is `__rascall`, which maps to the one that the authors of the library find most appropriate for a particular platform.

For C programs `bool` is `int`, `false` is 0, `true` is anything other than 0 (no specific value).

For C++ programs, function parameters marked with *OPTIONAL* can be omitted, in which case their value will be `NULL`, or zero. For C programs, this marker must be ignored and all parameters must be specified.

rascal_accept

```
rrid_t rascal_accept(
        const sock_t *,
        dispatcher,
        void *context OPTIONAL
);
```

Starts accepting incoming connections on the specified address. The identifier, if valid, can be used in a call to `rascal_cancel`. The `context` value is passed to the event dispatcher when time comes. Note that the value of the `context` parameter is inherited by all accepted connections; to identify a particular connection, you need to call `rascal_set_context` to change it when the connection is accepted.

rascal_accept_service

```
rrid_t rascal_accept_service(
        const char *name,
        const char *proto,
        const char *domain,
        dispatcher,
        void *context OPTIONAL
);
```

This function begins accepting connections to the specified service. It performs a service descovery DNS query, then attempts to install a listener on all listed addresses.

On success, this function installs multiple different listeners; the behaviour of this function can be resembled by a call to `rascal_connect_service` with a filter which calls `rascal_accept` then declines the outgoing connection attempt.

Parameters:

- `name`, `domain`, `proto` — service identification information. See `rascal_connect_service`.
- `dispatcher` — the address of an event dispatcher function. The `rop_listen` event will be delivered after each particular listener corresponding to the service is installed; the handler may return `false` to cancel a particular listener. The identifiers of listeners must be stored if a need to uninstall them may arise in future; the request identifier returned by `rascal_accept_service` is not suitable for that. Also, all further `rop_accept` notifications will carry the identifier of each particular listener, not the identifier returned by `rascal_accept_service` either.
- `context` — opaque user data passed back to the specified dispatcher when an event occurs.

When the return value indicates success, which means an operation is being performed in background, it may be used in conjunction with `rascal_cancel` to abort installing the listeners. It can not be used to uninstall them later.

Sample use:

```
rascal_accept_service("http", "tcp",
       "faerion.oss", my_dispatcher, NULL);
```

rascal_aton

```
bool rascal_aton(
       const char *symbolic,
       addr_t *addr,
       addr_t *mask OPTIONAL
);
```

Converts an address from its text representation to the numeric form. The string can be of one of the following forms:

1. Four octets of an IPv4 address in dotted quad notation. The `mask` parameter is set to 255.255.255.255.
2. An IPv4 address and a network mask in dotted quad notation, separated by a slash.
3. An IPv4 address and a network range (an integer number of significant higher bits of the address) separated by a slash.

The mask is ignored if the `mask` parameter is a null pointer or if the pattern does not contain a network mask.

If the address could be interpreted, the function returns `REC_SUCCESS` and `addr` and `mask` structures receive the corresponding data. If the `mask` parameter was not specified, the corresponding part of the source string is ignored.

When the function fails, the corresponding error message can be retreived by a call to `rascal_get_errmsg`. In that case, the content of `addr` and `mask` structures is undetermined.

rascal_cancel

```
rrid_t rascal_cancel(rrid_t);
```

Cancels a specific request. A failure is reported using whatever callback mechanism is supported by the request: either a `rop_close` event is delivered, or an address resolution failure is reported.

rascal_connect

Starts connectiong to a remote peer.

```
rrid_t rascal_connect(
        const sock_t *target,
        dispatcher,
        void *context OPTIONAL,
        const char *proto OPTIONAL
);
```

Parameters:

1. `target` — the address of the peer that the connection is to be established to.
2. `dispatcher` — address of the event dispatcher for this connection.
3. `context` — the value that helps the application identify the connection additionally to the request identifier assigned by this function (request identifiers are also persistent and do not change during connection's life time).
4. proto — type of the transport protocol that the connection must be established over. Currently, supported values are `proto_tcp` and `proto_udp`. With `proto_udp`, the connection will be established immediately (event dispatcher might be called before the function returns), and the application will be able to exchange datagrams with the remote peer.
   Note: the library does not take responsibility for the reliability of the communication process if the connection is made over `udp`; data may be lost without any notification.

If the connection was successfully scheduled (see `rascal_isok`), the return value can be used to interact with the connection, such as by reading or writing data with it. The connection can also be closed at any time by calling the `rascal_cancel` function.

When the function fails, a corresponding error message can be retreived using the `rascal_get_errmsg` function.

To establish a connection to a target identified by a symbolic host name, you need to either use the `rascal_getaddr` function or, alternatively, use the `rascal_connect_service` function. To connect to a system the numeric address of which is known, the following code would be sufficient:

```
sock_t peer(80);
rascal_aton("127.0.0.1", &peer.addr, NULL);
rascal_connect(&peer, NULL, dispatcher);
```

rascal_connect_service

```
rrid_t rascal_connect_service(
        const char *name,
        const char *proto,
        const char *domain,
        dispatcher,
        void *context OPTIONAL,
        filter &OPTIONAL;
);
```

This function attempts to connect to a service trying all available servers. Behaves similar to `rascal_connect`. For more information read about service discovery.

Parameters:

1. `service`, `domain` — service description. The name of the service, pointed by the `name` parameter, is the application level protocol, such as *http*, *ftp*, *smtp* or other defined by IANA or by common sense. The value of `domain` is the domain name to look for the service in. For details about service resolution refer to RFC2782.
2. `proto` — the type of the transport protocol to establish the connection over. Currently, only "tcp" and "udp" are supported by connection oriented functions.
   Note: the library does not take responsibility for the reliability of the communication process if the connection is made over `udp`; data may be lost without any notification.
3. dispatcher — the address of an event dispatcher function.
4. filter — the address of a function that filters the list of servers that provide the requested service. If this parameter is `NULL`, all service

providers are used.

The function must return true to allow the connection attempt to this server or false to skip it.

If the connection was successfully scheduled (see `rascal_isok`), the return value can be used to interact with the connection, such as by writing data to or reading from it. The connection can also be closed at any time by calling the `rascal_cancel`.

When the function fails, a corresponding error message can be retreived using the `rascal_get_errmsg` function.

The connection filter is:

```
bool __rascall filter(
        void *context,
        const char *host,
        const sock_t *addr
);
```

The function must return `true` to allow the connection attempt to this server or `false` to skip it. Parameters:

1. `context` — the value passed to `rascal_connect_service`, unaltered by the library.
2. `host`, `addr` — host name and numeric address of the server that the library will attemp to connect to, if the filter returns `true`.

The library may behave in two ways. Firstly, it may call the filter function right before it is about to connect to the next server in the list, typically after it failed to reach the previous server in the list. Or, it can first call the filter for each listed server to remove unwanted servers, and then start connecting. The application should not depend on each particular behaviour, because it may change.

Sample use:

```
rascal_connect_service(
        "http", "tcp", "faerion.oss",
        NULL, my_dispatcher, NULL);
```

rascal_dispatcher

This function receives notifications about all events raised by a connection established as a result of execution of functions `rascal_connect`, `rascal_connect_service` or `rascal_accept`. The return value is ignored, unless explicitly stated otherwise.

```
bool __rascall rascal_dispatcher(
        rrid_t conn,
        const sock_t *peer,
        int event,
        void *context
);
```

Parameters:

1. `conn` — request identifier. The application must test this value with the `rascal_isok` function to see if the operation failed or succeeded.
2. `peer` — the address of the remote connection endpoint. If the remote address is not known by the time the event is delivered, this parameter is filled with zeros.
3. `context` — the value passed to `rascal_connect` or other function that establishes a connection, unaltered by the library.

rascal_getaddr

Resolves a symbolic host name to a list of numeric addresses (A and AAAA record lookup).

```
rrid_t rascal_getaddr(
        const char *host,
        callback,
        void *context OPTIONAL
);
```

Results are delivered to a callback function of the following prototype:

```
void __rascall callback(
        void *context,
        const char *host,
        unsigned int count,
        const addr_t *addrs
);
```

If the value of `count` is zero, the lookup has failed, otherwise `addrs` points to an array of numeric addresses associated with the requested host name.

If the request could not be scheduled, the corresponding error message can be retreived using the `rascal_get_errmsg`. Otherwise, the return value can be used to cancel the request by a call to `rascal_cancel`; in that case, the callback function will be called by the library, with `count` set to zero.

rascal_gethost

Resolves the numeric address to a list of symbolic host names (PTR record lookup).

```
rrid_t rascal_gethost(
	const sock_t *addr,
	callback,
	void *context OPTIONAL
);
```

Results are delivered to a callback function of the following prototype:

```
void __rascall callback(
	void *context,
	const sock_t *addr,
	unsigned int count,
	const char **hosts
);
```

If the value of `count` is zero, then the lookup has failed, otherwise `hosts` points to an array of pointers to null-terminated host names associated with the requested address.

If the request could not be processed, the corresponding error message can be retreived using the `rascal_get_errmsg`. Otherwise, the return value can be used to cancel the request by a call to `rascal_cancel`; in that case, the callback function will be called by the library, having the `count` parameter set to zero.

rascal_get_rq_size

```
rrid_t rascal_get_rq_size(
	rrid_t,
	unsigned int *size
);
```

Use this to get the number of bytes in a connection's incoming buffer. If the function succeeds, `size` is updated with the size of the buffer. Note that for a datagram connection, size of all datagrams is returned, even though `rascal_read` will only return one datagram with each call.

rascal_get_sq_size

```
rrid_t rascal_get_sq_size(
	rrid_t,
	unsigned int *size
);
```

Use this to get the number of bytes in a connection's outgoing buffer. If the function succeeds, `size` is updated with the size of the queue.

rascal_get_errmsg

```
viod rascal_get_errmsg(
	rrid_t,
	char *buffer,
	unsigned int buffer_size
);
```

This function can be used to retreive a human readable error descriptions. The `rrid_t` parameter is the error code. `Buffer` and `buffer_size` define the location that receives the error description as a null-terminated string. If the buffer is not large enough, the message will be truncated.

Sample use:

```
if (!rascal_isok(rascal_do_something())) {
	char message[1024];
	rascal_get_errmsg(
		rid, message, sizeof(message));
	printf("Error: %s\n", message);
}
```

The messages are returned using the system default language, usually U.S. English.

rascal_init

```
rrid_t rascal_init(unsigned int policy);
```

This function initializes the library. Typically, it creates working threads and several threads for library's internal purposes. If initialization fails, the application should stop interacting with the library; all attempts to call functions that perform asynchronous actions will fail.

There is no function to shut the library down; this only happens when the process ceases.

The `policy` parameter defines the behaviour of the library and can be one of the following:

- `RIP_WORKER_MANUAL`, tells the library not to spawn worker threads. The application must periodically call `rascal_work` to perform all outstanding tasks and dispatch events. This mode is useful for applications that have a running loop and want everything — both the non-library code and event dispatching — happen within a single thread.
- `RIP_WORKER_SINGLE`, tells the library to create a single worker thread that dispatches events "in the background". This mode is useful for applications that serve a GUI in the main thread, such as applications written with MFC or other libraries, and the developer does not want to hack into the main program loop to call `rascal_work` from there. This is the recommended mode of execution for most GUI applications and applications that don't fight for best performance.
- `RIP_WORKER_PER_CPU`, tells the library to launch as many worker threads as there are CPUs installed. On systems that support it, affinity masks are set so that each thread runs on a different CPU. This mode is best for high-performance applications.

In a multithreaded environment the application must use its own means to protect the data from being accessed simultaenously by multiple threads.

rascal_isok

Checks if the result identifier indicates an error.

```
bool rascal_isok(rrid_t id);
```

This function returns `true` if the operation identified by `id` has been executed without errors. If `false` is returned, `id` can be fed to the `rascal_get_errmsg` function to retreive the corresponding error message.

rascal_ntoa

```
void rascal_ntoa(
        const addr_t *source,
        char *buffer,
        unsigned int buffer_size
);
```

Fills the buffer with a null-terminated string that represents the given address in a text form in the most common representation.

rascal_read

This function retreives data from the connection's incoming buffer.

```
rrid_t rascal_read(
        rrid_t conn,
        void *buffer,
        unsigned int *buffer_size
);
```

Parameters:

1. `conn` — handle of the connection that the data must be retreived from. If the handle does not correspond to a valid data oriented connection, an error is reported.
2. `buffer`, `buffer_size` — location and dimension of the buffer that receives the data. Before calling the function, `buffer_size` must be set to the size of the buffer, in bytes. When the function returns with success, `buffer_size` is updated with the number of bytes actually retreived; this value never exceeds the original buffer size.

On success, the function returns `REC_SUCCESS`, otherwise an error code is returned. Error code `REC_NO_DATA` means that the buffer was empty and to data could be retreived (whether this is an error or not is up to the application).

When reading from a stream-oriented connection, the whole incoming buffer can be retreived with one call. If `conn` corresponds to a datagram connection, only one full datagram will be retreived with each call, no matter how large the buffer is. When no more datagrams are available, the function returns `REC_NO_DATA`.

rascal_reads

This function retreives a line of text from the connection's incoming buffer. A line of text is a sequence of 8-bit characters terminated by an LF character.

```
rrid_t rascal_reads(
        rrid_t conn,
        char *buffer,
        unsigned int *buffer_size,
        int flags OPTIONAL
);
```

Parameters:

1. `conn` — handle of the connection that the data must be retreived from. If the handle does not correspond to a valid stream oriented connection, an error is reported. This function can not be used on datagram connections.
2. `buffer`, `buffer_size` — location and dimension of the buffer that receives the a NUL-terminated C-string. On success, `buffer_size` is updated with the number of actually copied bytes, excluding the (trailing) NUL character. (That is, the maximum possible value of `buffer_size` on return is the initial value minus one.)
3. `flags` — a bit mask that modifies the behaviour of the function. By default (when zero is used for `flags`), the function does the following:
    1. Copies the data to the specified destination, excluding all trailing CR and LF characters. If the destination is not large enough, the data is truncated.
    2. Removes the data from the connection's incoming buffer, up to and including the LF character.

    A combination of the following flags can be used to modify this behaviour:

    - `RRF_UNTRIMMED`, tells the function to include trailing CR and LF characters in the destination buffer.
    - `RRF_PEEK`, prevents the function from removing the data from the connection's incoming buffer; the data is copied to the specified destination and remains in the buffer.
    - `RRF_MEASURE`, prevents the function from extracting anything; instead, `size` is updated with the length of the buffer that would be enough to receive the whole string, without truncation. Other flags that modify the copied string (such as `RRF_UNTRIMMED`) are taken into consideration.

If everything goes well, the function returns `REC_SUCCESS`. If the connection's incoming buffer did not contain an LF character, the function returns `REC_NO_DATA`. Descriptions of other error messages can be retreived using `rascal_get_errmsg`.

Sample use:

```
char buffer[1024];
unsigned int size = sizeof(buffer);
while (rascal_isok(rascal_reads(rid, buffer,
&size))) {
        printf(">> %s\n", buffer);
        size = sizeof(buffer);
}
```

rascal_set_context

```
rrid_t rascal_set_context(
        rrid_t,
        void *context
);
```

Modifies the context identifier for a connection. If `conn` does not correspond to a data oriented connection, an error is reported.

rascal_wait

```
rrid_t rasacl_wait(rrid_t  id);
```

Suspends the current thread until the request with the specified `id` is finished, either successfully or with an error. This function is for applications that do not need to work asynchronously (such as simple command line utilities); it lets avoid using semaphores and other signalisation.

rascal_work

```
bool rascal_work(unsigned int msec);
```

Lets the library perform pending operations and dispatch the events. The library will spend `msec` milliseconds waiting for an operation to complete and then return without doing anything. If there was job to do, the library will most likely to return before the specified amount of time has elapsed.

Using this function makes sense only if the library was started in the `RIP_WORKER_MANUAL` mode; in all other modes the library already has at least one worker thread that calls this function in an infinite loop.

The function returns `true` if there were events dispatched, `false` if nothing was done.

rascal_write

Writes data to a connection.

```
rrid_t rascal_write(
        rrid_t conn,
        const char *buffer,
        unsigned int buffer_size
);
```

Parameters:

1. `conn` — identifier of the connection that the data must be retreived from. If the handle does not correspond to a valid data oriented connection, an error is reported.
2. `buffer`, `buffer_size` — location and size of the block of data must be sent. The function copies the data to the connection's internal buffer, thus `buffer` may point to stack-based data.

   For datagram connections, one call to `rascal_write` adds one datagram to the outgoing buffer. The library truncates datagrams to 2048 bytes; the underlying transport provider may truncate it to an even smaller size. The common size limit for a datagram is 512 bytes.

   For stream oriented connections, there is no limit on the size of the data.

On success, the function returns `REC_SUCCESS`. Otherwise, the corresponding error message can be retreived with `rascal_get_errmsg`. If the function fails, it does not necessarily mean that the connection is no longer usable; instead, the application should wait for a `rop_close` event to prove that.

Advanced functions

This section describes advanced functions.

rascal_get_option

This function retreives values of options that control various aspects of the library's internal functioning.

```
rrid_t rascal_get_option(
        unsigned int optid,
        long int *value
);
```

Parameters:

1. `optid` — identifies the option that the application wants to retrieve the value of. For a list of available options, refer to the `rascal_set_option` function description.
2. `value` — if not null, receives the current value of the option.

On success, the function returns `REC_SUCCESS`. Otherwise, an error code is returned which can then be used in a call to `rascal_get_errmsg`.

rascal_set_dispatcher

```
rrid_t rascal_set_dispatcher(
        rrid_t,
        rascal_dispatcher new_dispatcher
);
```

Changes the address of the dispatcher

function for a request.

rascal_set_nameserver

```
void rascal_set_nameserver(
        const sock_t *list,
        unsigned int count
);
```

Parameters:

1. `list`, `count` — the array of addresses of name servers to be used.

This function opens connections to a number of name servers. When more than one server is specified; each query is sent to all servers, the first response is used. This is usually used as a measure against DNS server failure. Currently, the library supports up to 16 name servers.

The library connects to all name servers defined in the system configuration during start-up; you don't need to use this function unless you want to access non-default name servers. (For example, when you want to use OpenNIC on systems that you can not reconfigure.)

rascal_set_option

This function can be used for fine-tuning the library. Fine-tuning is done by changing so-called "options", each of which is has a 32-bit signed integer value assigned to it.

```
rrid_t rascal_set_option(
        unsigned int optid,
        long int value,
        long int *old_value OPTIONAL
);
```

Parameters:

- `optid` — identifies the option that the application wants changed. Possible options are:
  - `RO_VOID` — no meaning, holds a user-defined 32-bit value.
  - `RO_DNS_TIMEOUT` — the number of milliseconds for a DNS query to remain unanswered before being resent.
  - `RO_DNS_RETRY` — the number of times an unanswered DNS query is resent before a failure is reported. Setting this to zero will disable retries, having the queries only sent once.
  - `RO_THREAD_POLICY` — threading policy used by the library. The value can only be set once and is set by the `rascal_init` function. Once the option is set, it becomes read-only. If you change this option before initializing the library with a call to `rascal_init`, the library will fail to initialize and normal execution will not be possible.
  - `RO_CONN_TIMEOUT` — outgoing connection timeout, in seconds. Connection attempts that do not finish during the specified time period will fail with `REC_CONN_TIMEOUT`.
- `value` — the new value for the option. The library performs no sanity check of the values.
- `old_value` — if not null, receives the old value of the option.

On success, the function returns `REC_SUCCESS`. If the option can not be changed, `REC_OPTION_READONLY` is returned. Otherwise, an error code is returned which can then be used in a call to `rascal_get_errmsg`.

The options are process-wide. The changes are immediately visible to all threads.

rascal_shrink

```
void rascal_shrink(void);
```

Releases all internally cached resources that are currently not being used. This function should not be called often, because that would cause performance penalty: the memory pool is constantly reused without the implication of memory allocation functions which saves time on thread safety (memory heap locking) and so on.

Internal reference

The intended audience of this section is the developers willing to port the library onto a yet unsupported platform. Regular users may ignore this section for they will never deal with anything described in it. The mission of this section is to describe all resources available for use. Re-miplementing functions covered by the described classes is not recommended.

datachain

This class implements data stream storage in fragments of a fixed small size (about 2048 bytes each). The class provides means for copying data from the data chain into a linear buffer and extracting EOL-terminated lines of text. The class uses paged memory allocation and is efficient when it comes to chain extension or truncation. Thread locking is provided internally by the class.

The `datachain` class is a header that manages a number of linked `datachainitem` objects. Each object can hold 2048 bytes of data and information of how many bytes are `used` in each item and how many are `dead`; the latter means the bytes that are already processed and should be discarded (this is to avoid moving data as it is peeked from the chain). The amount of `dead` bytes never exceeds the number of `used` bytes.

```
void ext_used(char *&, unsigned int &);
```

This method finds the first element in which not all bytes are used and returns a pointer to the first unused byte and the number of unused bytes in the chain element. If there is no element with unused bytes, a new one is allocated. This opeartion is typically performed when reading needs to be done: you request a space to read to, then fix it with a call to `add_used`.

```
void add_used(unsigned int);
```

This method marks the specified number of unused bytes as used. Typically this is done after requesting unused space by a call to `ext_used` to "fix" the data after it was copied into the buffer, thus making it actually visible and available.

The `datachain` class tracks the chain element that was last extended with a call to `ext_used` and, until a subsequent call to `add_used` follows, all further calls to `ext_used` will fail by returning a zero sized buffer.

pageman

This is the memory allocation class that supports paged object allocation to, basically, decrease memory fragmentation. This class should be used for objects more or less frequently allocated not on stack. Typical use for this class:

```
#include "pageman.h"

class myclass {
        // 100 is the size of a page
        pageman pmgr<myclass>(100);
public:
        void* operator new
                (unsigned int size)
                { return pmgr.alloc(); }
        void operator delete
                (void *object)
                { pmgr.free(object); }
};
```

The content of objects allocated this way is not initialized; neither constructor nor destructor are called for objects managed by the page manager.

Overrunning the borders of an allocated object by at least one byte will lead to fatal damage during following operation on that pageman object.

There is a helper `CACHED_ALLOCATORS` macro which receives the name of a `pageman` object and declares operators `new` and `delete`. When used, the above class declaration is simplified up to the following:

```
class myclass {
       pageman pmgr<myclass>(100);
public:
       CACHED_ALLOCATORS(pmgr);
};
```

stock

A protected container class used to hide pointers from the client application. Maps pointers to unique identifiers. Detects modified identifiers and disallows them from use (to prevent memory access violation). Defined in `stock.h`, not defined in standard header.

Memory allocation is paged, page size is customizable and defaults to 100. Object list access is very fast and involves no array scan. The only exception is the allocation of a new page, which involves initialization of an array of 100 integers.

Objects stored inside a `stock` class must be fully relocatable. Never store pointers to these objects, they may become invalid after container extension. Typically, you will only want to store pointers in a `stock` class.

```
int stock<T>::add(T obj);
```

Inserts an object into the container. Currently, the class can store up to 65536 objects. Inserting more objects will result in an error indicated by a negative result value. The same negative value is returned if a memory allocation error occurs. Otherwise a positive object index is returned; zero is a valid index.

```
bool stock<T>::get(int idx, T& obj);
```

Extracts an object from the container. Returns `true` on success, `false` if the object could not be found.

```
bool stock<T>::remove(int idx);
```

Removes an object from the list. Returns `true` if the object was successfully removed, `false` if there was no such object. Removing an object does not involve memory allocation; the slot is marked available for reuse. The id of a removed object becomes invalid and all further operations on it will fail.