[Build Status](https://travis-ci.org/redis/hiredis)

**This Readme reflects the latest changed in the master branch. See** [**v0.13.3**](https://github.com/redis/hiredis/tree/v0.13.3) **for the Readme and documentation for the latest release.**

# HIREDIS

Hiredis is a minimalistic C client library for the [Redis](http://redis.io/) database.

It is minimalistic because it just adds minimal support for the protocol, but at the same time it uses a high level printf-alike API in order to make it much higher level than otherwise suggested by its minimal code base and the lack of explicit bindings for every Redis command.

Apart from supporting sending commands and receiving replies, it comes with a reply parser that is decoupled from the I/O layer. It is a stream parser designed for easy reusability, which can for instance be used in higher level language bindings for efficient reply parsing.

Hiredis only supports the binary-safe Redis protocol, so you can use it with any Redis version >= 1.2.0.

The library comes with multiple APIs. There is the *synchronous API*, the *asynchronous API* and the *reply parsing API*.

## Upgrading to 1.0.0

Version 1.0.0 marks a stable release of hiredis. It includes some minor breaking changes, mostly to make the exposed API more uniform and self-explanatory. It also bundles the updated sds library, to sync up with upstream and Redis. For most applications a recompile against the new hiredis should be enough. For code changes see the [Changelog](http://changelog.md).

## Upgrading from <0.9.0

Version 0.9.0 is a major overhaul of hiredis in every aspect. However, upgrading existing code using hiredis should not be a big pain. The key thing to keep in mind when upgrading is that hiredis >= 0.9.0 uses a redisContext\* to keep state, in contrast to the stateless 0.0.1 that only has a file descriptor to work with.

## Synchronous API

To consume the synchronous API, there are only a few function calls that need to be introduced:

redisContext \*redisConnect(const char \*ip, int port);

void \*redisCommand(redisContext \*c, const char \*format, ...);

void freeReplyObject(void \*reply);

### Connecting

The function redisConnect is used to create a so-called redisContext. The context is where Hiredis holds state for a connection. The redisContext struct has an integer err field that is non-zero when the connection is in an error state. The field errstr will contain a string with a description of the error. More information on errors can be found in the **Errors** section. After trying to connect to Redis using redisConnect you should check the err field to see if establishing the connection was successful:

redisContext \*c = redisConnect("127.0.0.1", 6379);

if (c == NULL || c->err) {

if (c) {

printf("Error: %s\n", c->errstr);

// handle error

} else {

printf("Can't allocate redis context\n");

}

}



*Note: A redisContext is not thread-safe.*

### Sending commands

There are several ways to issue commands to Redis. The first that will be introduced is redisCommand. This function takes a format similar to printf. In the simplest form, it is used like this:

reply = redisCommand(context, "SET foo bar");



The specifier %s interpolates a string in the command, and uses strlen to determine the length of the string:

reply = redisCommand(context, "SET foo %s", value);



When you need to pass binary safe strings in a command, the %b specifier can be used. Together with a pointer to the string, it requires a size\_t length argument of the string:

reply = redisCommand(context, "SET foo %b", value, (size\_t) valuelen);



Internally, Hiredis splits the command in different arguments and will convert it to the protocol used to communicate with Redis. One or more spaces separates arguments, so you can use the specifiers anywhere in an argument:

reply = redisCommand(context, "SET key:%s %s", myid, value);

### Using replies

The return value of redisCommand holds a reply when the command was successfully executed. When an error occurs, the return value is NULL and the err field in the context will be set (see section on **Errors**). Once an error is returned the context cannot be reused and you should set up a new connection.

The standard replies that redisCommand are of the type redisReply. The type field in the redisReply should be used to test what kind of reply was received:

* **REDIS\_REPLY\_STATUS**:
  + The command replied with a status reply. The status string can be accessed using reply->str. The length of this string can be accessed using reply->len.
* **REDIS\_REPLY\_ERROR**:
  + The command replied with an error. The error string can be accessed identical to REDIS\_REPLY\_STATUS.
* **REDIS\_REPLY\_INTEGER**:
  + The command replied with an integer. The integer value can be accessed using the reply->integer field of type long long.
* **REDIS\_REPLY\_NIL**:
  + The command replied with a **nil** object. There is no data to access.
* **REDIS\_REPLY\_STRING**:
  + A bulk (string) reply. The value of the reply can be accessed using reply->str. The length of this string can be accessed using reply->len.
* **REDIS\_REPLY\_ARRAY**:
  + A multi bulk reply. The number of elements in the multi bulk reply is stored in reply->elements. Every element in the multi bulk reply is a redisReply object as well and can be accessed via reply->element[..index..]. Redis may reply with nested arrays but this is fully supported.

Replies should be freed using the freeReplyObject() function. Note that this function will take care of freeing sub-reply objects contained in arrays and nested arrays, so there is no need for the user to free the sub replies (it is actually harmful and will corrupt the memory).

**Important:** the current version of hiredis (0.10.0) frees replies when the asynchronous API is used. This means you should not call freeReplyObject when you use this API. The reply is cleaned up by hiredis *after* the callback returns. This behavior will probably change in future releases, so make sure to keep an eye on the changelog when upgrading (see issue #39).

### Cleaning up

To disconnect and free the context the following function can be used:

void redisFree(redisContext \*c);



This function immediately closes the socket and then frees the allocations done in creating the context.

### Sending commands (cont'd)

Together with redisCommand, the function redisCommandArgv can be used to issue commands. It has the following prototype:

void \*redisCommandArgv(redisContext \*c, int argc, const char \*\*argv, const size\_t \*argvlen);



It takes the number of arguments argc, an array of strings argv and the lengths of the arguments argvlen. For convenience, argvlen may be set to NULL and the function will use strlen(3) on every argument to determine its length. Obviously, when any of the arguments need to be binary safe, the entire array of lengths argvlen should be provided.

The return value has the same semantic as redisCommand.

### Pipelining

To explain how Hiredis supports pipelining in a blocking connection, there needs to be understanding of the internal execution flow.

When any of the functions in the redisCommand family is called, Hiredis first formats the command according to the Redis protocol. The formatted command is then put in the output buffer of the context. This output buffer is dynamic, so it can hold any number of commands. After the command is put in the output buffer, redisGetReply is called. This function has the following two execution paths:

1. The input buffer is non-empty:
   * Try to parse a single reply from the input buffer and return it
   * If no reply could be parsed, continue at *2*
2. The input buffer is empty:
   * Write the **entire** output buffer to the socket
   * Read from the socket until a single reply could be parsed

The function redisGetReply is exported as part of the Hiredis API and can be used when a reply is expected on the socket. To pipeline commands, the only things that needs to be done is filling up the output buffer. For this cause, two commands can be used that are identical to the redisCommand family, apart from not returning a reply:

void redisAppendCommand(redisContext \*c, const char \*format, ...);

void redisAppendCommandArgv(redisContext \*c, int argc, const char \*\*argv, const size\_t \*argvlen);



After calling either function one or more times, redisGetReply can be used to receive the subsequent replies. The return value for this function is either REDIS\_OK or REDIS\_ERR, where the latter means an error occurred while reading a reply. Just as with the other commands, the err field in the context can be used to find out what the cause of this error is.

The following examples shows a simple pipeline (resulting in only a single call to write(2) and a single call to read(2)):

redisReply \*reply;

redisAppendCommand(context,"SET foo bar");

redisAppendCommand(context,"GET foo");

redisGetReply(context,&reply); // reply for SET

freeReplyObject(reply);

redisGetReply(context,&reply); // reply for GET

freeReplyObject(reply);



This API can also be used to implement a blocking subscriber:

reply = redisCommand(context,"SUBSCRIBE foo");

freeReplyObject(reply);

while(redisGetReply(context,&reply) == REDIS\_OK) {

// consume message

freeReplyObject(reply);

}

### Errors

When a function call is not successful, depending on the function either NULL or REDIS\_ERR is returned. The err field inside the context will be non-zero and set to one of the following constants:

* **REDIS\_ERR\_IO**: There was an I/O error while creating the connection, trying to write to the socket or read from the socket. If you included errno.h in your application, you can use the global errno variable to find out what is wrong.
* **REDIS\_ERR\_EOF**: The server closed the connection which resulted in an empty read.
* **REDIS\_ERR\_PROTOCOL**: There was an error while parsing the protocol.
* **REDIS\_ERR\_OTHER**: Any other error. Currently, it is only used when a specified hostname to connect to cannot be resolved.

In every case, the errstr field in the context will be set to hold a string representation of the error.

## Asynchronous API

Hiredis comes with an asynchronous API that works easily with any event library. Examples are bundled that show using Hiredis with [libev](http://software.schmorp.de/pkg/libev.html) and [libevent](http://monkey.org/~provos/libevent/).

### Connecting

The function redisAsyncConnect can be used to establish a non-blocking connection to Redis. It returns a pointer to the newly created redisAsyncContext struct. The err field should be checked after creation to see if there were errors creating the connection. Because the connection that will be created is non-blocking, the kernel is not able to instantly return if the specified host and port is able to accept a connection.

*Note: A redisAsyncContext is not thread-safe.*

redisAsyncContext \*c = redisAsyncConnect("127.0.0.1", 6379);

if (c->err) {

printf("Error: %s\n", c->errstr);

// handle error

}



The asynchronous context can hold a disconnect callback function that is called when the connection is disconnected (either because of an error or per user request). This function should have the following prototype:

void(const redisAsyncContext \*c, int status);



On a disconnect, the status argument is set to REDIS\_OK when disconnection was initiated by the user, or REDIS\_ERR when the disconnection was caused by an error. When it is REDIS\_ERR, the err field in the context can be accessed to find out the cause of the error.

The context object is always freed after the disconnect callback fired. When a reconnect is needed, the disconnect callback is a good point to do so.

Setting the disconnect callback can only be done once per context. For subsequent calls it will return REDIS\_ERR. The function to set the disconnect callback has the following prototype:

int redisAsyncSetDisconnectCallback(redisAsyncContext \*ac, redisDisconnectCallback \*fn);

### Sending commands and their callbacks

In an asynchronous context, commands are automatically pipelined due to the nature of an event loop. Therefore, unlike the synchronous API, there is only a single way to send commands. Because commands are sent to Redis asynchronously, issuing a command requires a callback function that is called when the reply is received. Reply callbacks should have the following prototype:

void(redisAsyncContext \*c, void \*reply, void \*privdata);



The privdata argument can be used to curry arbitrary data to the callback from the point where the command is initially queued for execution.

The functions that can be used to issue commands in an asynchronous context are:

int redisAsyncCommand(

redisAsyncContext \*ac, redisCallbackFn \*fn, void \*privdata,

const char \*format, ...);

int redisAsyncCommandArgv(

redisAsyncContext \*ac, redisCallbackFn \*fn, void \*privdata,

int argc, const char \*\*argv, const size\_t \*argvlen);



Both functions work like their blocking counterparts. The return value is REDIS\_OK when the command was successfully added to the output buffer and REDIS\_ERR otherwise. Example: when the connection is being disconnected per user-request, no new commands may be added to the output buffer and REDIS\_ERR is returned on calls to the redisAsyncCommand family.

If the reply for a command with a NULL callback is read, it is immediately freed. When the callback for a command is non-NULL, the memory is freed immediately following the callback: the reply is only valid for the duration of the callback.

All pending callbacks are called with a NULL reply when the context encountered an error.

### Disconnecting

An asynchronous connection can be terminated using:

void redisAsyncDisconnect(redisAsyncContext \*ac);



When this function is called, the connection is **not** immediately terminated. Instead, new commands are no longer accepted and the connection is only terminated when all pending commands have been written to the socket, their respective replies have been read and their respective callbacks have been executed. After this, the disconnection callback is executed with the REDIS\_OK status and the context object is freed.

### Hooking it up to event library *X*

There are a few hooks that need to be set on the context object after it is created. See the adapters/ directory for bindings to *libev* and *libevent*.

## Reply parsing API

Hiredis comes with a reply parsing API that makes it easy for writing higher level language bindings.

The reply parsing API consists of the following functions:

redisReader \*redisReaderCreate(void);

void redisReaderFree(redisReader \*reader);

int redisReaderFeed(redisReader \*reader, const char \*buf, size\_t len);

int redisReaderGetReply(redisReader \*reader, void \*\*reply);



The same set of functions are used internally by hiredis when creating a normal Redis context, the above API just exposes it to the user for a direct usage.

### Usage

The function redisReaderCreate creates a redisReader structure that holds a buffer with unparsed data and state for the protocol parser.

Incoming data -- most likely from a socket -- can be placed in the internal buffer of the redisReader using redisReaderFeed. This function will make a copy of the buffer pointed to by buf for len bytes. This data is parsed when redisReaderGetReply is called. This function returns an integer status and a reply object (as described above) via void \*\*reply. The returned status can be either REDIS\_OK or REDIS\_ERR, where the latter means something went wrong (either a protocol error, or an out of memory error).

The parser limits the level of nesting for multi bulk payloads to 7. If the multi bulk nesting level is higher than this, the parser returns an error.

### Customizing replies

The function redisReaderGetReply creates redisReply and makes the function argument reply point to the created redisReply variable. For instance, if the response of type REDIS\_REPLY\_STATUS then the str field of redisReply will hold the status as a vanilla C string. However, the functions that are responsible for creating instances of the redisReply can be customized by setting the fn field on the redisReader struct. This should be done immediately after creating the redisReader.

For example, [hiredis-rb](https://github.com/pietern/hiredis-rb/blob/master/ext/hiredis_ext/reader.c) uses customized reply object functions to create Ruby objects.

### Reader max buffer

Both when using the Reader API directly or when using it indirectly via a normal Redis context, the redisReader structure uses a buffer in order to accumulate data from the server. Usually this buffer is destroyed when it is empty and is larger than 16 KiB in order to avoid wasting memory in unused buffers

However when working with very big payloads destroying the buffer may slow down performances considerably, so it is possible to modify the max size of an idle buffer changing the value of the maxbuf field of the reader structure to the desired value. The special value of 0 means that there is no maximum value for an idle buffer, so the buffer will never get freed.

For instance if you have a normal Redis context you can set the maximum idle buffer to zero (unlimited) just with:

context->reader->maxbuf = 0;



This should be done only in order to maximize performances when working with large payloads. The context should be set back to REDIS\_READER\_MAX\_BUF again as soon as possible in order to prevent allocation of useless memory.

## AUTHORS

Hiredis was written by Salvatore Sanfilippo (antirez at gmail) and Pieter Noordhuis (pcnoordhuis at gmail) and is released under the BSD license.  
Hiredis is currently maintained by Matt Stancliff (matt at genges dot com) and Jan-Erik Rediger (janerik at fnordig dot com)