Redis Protocol specification

Redis clients communicate with the Redis server using a protocol called **RESP** (REdis Serialization Protocol). While the protocol was designed specifically for Redis, it can be used for other client-server software projects.

RESP is a compromise between the following things:

- Simple to implement.
- Fast to parse.
- Human readable.

RESP can serialize different data types like integers, strings, arrays. There is also a specific type for errors. Requests are sent from the client to the Redis server as arrays of strings representing the arguments of the command to execute. Redis replies with a command-specific data type.

RESP is binary-safe and does not require processing of bulk data transferred from one process to another, because it uses prefixed-length to transfer bulk data.

Note: the protocol outlined here is only used for client-server communication. Redis Cluster uses a different binary protocol in order to exchange messages between nodes.

Networking layer

A client connects to a Redis server creating a TCP connection to the port 6379.

While RESP is technically non-TCP specific, in the context of Redis the protocol is only used with TCP connections (or equivalent stream oriented connections like Unix sockets).

Request-Response model

Redis accepts commands composed of different arguments. Once a command is received, it is processed and a reply is sent back to the client.

This is the simplest model possible, however there are two exceptions:

- Redis supports pipelining (covered later in this document). So it is possible for clients to send multiple commands at once, and wait for replies later.
- When a Redis client subscribes to a Pub/Sub channel, the protocol changes semantics and becomes a *push* protocol, that is, the client no longer requires to send commands, because the server will automatically send to the client new messages (for the channels the client is subscribed to) as soon as they are received.

Excluding the above two exceptions, the Redis protocol is a simple request-response protocol.

RESP protocol description

The RESP protocol was introduced in Redis 1.2, but it became the standard way for talking with the Redis server in Redis 2.0. This is the protocol you should implement in your Redis client.

RESP is actually a serialization protocol that supports the following data types: Simple Strings, Errors, Integers, Bulk Strings and Arrays.

The way RESP is used in Redis as a request-response protocol is the following:

- Clients send commands to a Redis server as a RESP Array of Bulk Strings.
- The server replies with one of the RESP types according to the command implementation.

In RESP, the type of some data depends on the first byte:

- For **Simple Strings** the first byte of the reply is "+"
- For **Errors** the first byte of the reply is "-"
- For **Integers** the first byte of the reply is ":"
- For **Bulk Strings** the first byte of the reply is "$"
- For **Arrays** the first byte of the reply is "*"

Additionally RESP is able to represent a Null value using a special variation of Bulk Strings or Array as specified later.

In RESP different parts of the protocol are always terminated with "\r\n" (CRLF).

RESP Simple Strings

Simple Strings are encoded in the following way: a plus character, followed by a string that cannot contain a CR or LF character (no newlines are allowed), terminated by CRLF (that is "\r\n").

Simple Strings are used to transmit non binary safe strings with minimal overhead. For example many Redis commands reply with just "OK" on success, that as a RESP Simple String is encoded with the following 5 bytes:

```
"+OK\r\n"
```

In order to send binary-safe strings, RESP Bulk Strings are used instead.

When Redis replies with a Simple String, a client library should return to the caller a string composed of the first character after the '+' up to the end of the string, excluding the final CRLF bytes.

RESP Errors

RESP has a specific data type for errors. Actually errors are exactly like RESP Simple Strings, but the first character is a minus '-' character instead of a plus. The real difference between Simple Strings and Errors in RESP is that errors are treated by clients as exceptions, and the string that composes the Error type is the error message itself.

The basic format is:

```
"-Error message\r\n"
```

Error replies are only sent when something wrong happens, for instance if you try to perform an operation against the wrong data type, or if the command does not exist and so forth. An exception should be raised by the library client when an Error Reply is received.

The following are examples of error replies:

```
-ERR unknown command 'foobar'
-WRONGTYPE Operation against a key holding the wrong kind of value
```

The first word after the "-", up to the first space or newline, represents the kind of error returned. This is just a convention used by Redis and is not part of the RESP Error format.

For example, ERR is the generic error, while `WRONGTYPE` is a more specific error that implies that the client tried to perform an operation against the wrong data type. This is called an **Error Prefix** and is a way to allow the client to understand the kind of error returned by the server without to rely on the exact message given, that may change over the time.

A client implementation may return different kind of exceptions for different errors, or may provide a generic way to trap errors by directly providing the error name to the caller as a string.

However, such a feature should not be considered vital as it is rarely useful, and a limited client implementation may simply return a generic error condition, such as `false`.

RESP Integers

This type is just a CRLF terminated string representing an integer, prefixed by a ":" byte. For example ":0\r\n", or ":1000\r\n" are integer replies.

Many Redis commands return RESP Integers, like INCR, LLEN and LASTSAVE.

There is no special meaning for the returned integer, it is just an incremental number for INCR, a UNIX time for LASTSAVE and so forth. However, the returned integer is guaranteed to be in the range of a signed 64 bit integer.

Integer replies are also extensively used in order to return true or false. For instance commands like EXISTS or SISMEMBER will return 1 for true and 0 for false.

Other commands like SADD, SREM and SETNX will return 1 if the operation was actually performed, 0 otherwise.

The following commands will reply with an integer reply: SETNX, DEL, EXISTS, INCR, INCRBY, DECR, DECRBY, DBSIZE, LASTSAVE, RENAMENX, MOVE, LLEN, SADD, SREM, SISMEMBER, SCARD.

RESP Bulk Strings

Bulk Strings are used in order to represent a single binary safe string up to 512 MB in length.

Bulk Strings are encoded in the following way:

- A "$" byte followed by the number of bytes composing the string (a prefixed length), terminated by CRLF.
- The actual string data.
- A final CRLF.

So the string "foobar" is encoded as follows:

```
"$6\r\nfoobar\r\n"
```

When an empty string is just:

```
"$0\r\n\r\n"
```

RESP Bulk Strings can also be used in order to signal non-existence of a value using a special format that is used to represent a Null value. In this special format the length is -1, and there is no data, so a Null is represented as:

```
"$-1\r\n"
```

This is called a **Null Bulk String**.

The client library API should not return an empty string, but a nil object, when the server replies with a Null Bulk String. For example a Ruby library should return 'nil' while a C library should return NULL (or set a special flag in the reply object), and so forth.

RESP Arrays

Clients send commands to the Redis server using RESP Arrays. Similarly certain Redis commands returning collections of elements to the client use RESP Arrays are reply type. An example is the LRANGE command that returns elements of a list.

RESP Arrays are sent using the following format:

- A `*` character as the first byte, followed by the number of elements in the array as a decimal number, followed by CRLF.
- An additional RESP type for every element of the Array.

So an empty Array is just the following:

```
"*0\r\n"
```

While an array of two RESP Bulk Strings "foo" and "bar" is encoded as:

```
"*2\r\n$3\r\nfoo\r\n$3\r\nbar\r\n"
```

As you can see after the `*<count>CRLF` part prefixing the array, the other data types composing the array are just concatenated one after the other. For example an Array of three integers is encoded as follows:

```
"*3\r\n:1\r\n:2\r\n:3\r\n"
```

Arrays can contain mixed types, it's not necessary for the elements to be of the same type. For instance, a list of four integers and a bulk string can be encoded as the follows:

```
*5\r\n
:1\r\n
:2\r\n
:3\r\n
:4\r\n
$6\r\n
foobar\r\n
```

(The reply was split into multiple lines for clarity).

The first line the server sent is `*5\r\n` in order to specify that five replies will follow. Then every reply constituting the items of the Multi Bulk reply are transmitted.

The concept of Null Array exists as well, and is an alternative way to specify a Null value (usually the Null Bulk String is used, but for historical reasons we have two formats).

For instance when the BLPOP command times out, it returns a Null Array that has a count of -1 as in the following example:

```
"*-1\r\n"
```

A client library API should return a null object and not an empty Array when Redis replies with a Null Array. This is necessary to distinguish between an empty list and a different condition (for instance the timeout condition of the BLPOP command).

Arrays of arrays are possible in RESP. For example an array of two arrays is encoded as follows:

```
*2\r\n
*3\r\n
:1\r\n
:2\r\n
:3\r\n
*2\r\n
+Foo\r\n
-Bar\r\n
```

(The format was split into multiple lines to make it easier to read).

The above RESP data type encodes a two elements Array consisting of an Array that contains three Integers 1, 2, 3 and an array of a Simple String and an Error.

Null elements in Arrays

Single elements of an Array may be Null. This is used in Redis replies in order to signal that this elements are missing and not empty strings. This can happen with the SORT command when used with the GET *pattern* option when the specified key is missing. Example of an Array reply containing a Null element:

```
*3\r\n
$3\r\n
foo\r\n
$-1\r\n
$3\r\n
bar\r\n
```

The second element is a Null. The client library should return something like this:

```
["foo",nil,"bar"]
```

Note that this is not an exception to what said in the previous sections, but just an example to further specify the protocol.

Sending commands to a Redis Server

Now that you are familiar with the RESP serialization format, writing an implementation of a Redis client library will be easy. We can further specify how the interaction between the client and the server works:

- A client sends to the Redis server a RESP Array consisting of just Bulk Strings.
- A Redis server replies to clients sending any valid RESP data type as reply.

So for example a typical interaction could be the following.

The client sends the command **LLEN mylist** in order to get the length of the list stored at key *mylist*, and the server replies with an Integer reply as in the following example (C: is the client, S: the server).

```
C: *2\r\n
C: $4\r\n
C: LLEN\r\n
C: $6\r\n
C: mylist\r\n

S: :48293\r\n
```

As usually we separate different parts of the protocol with newlines for simplicity, but the actual interaction is the client sending `*2\r\n$4\r\nLLEN\r\n$6\r\nmylist\r\n` as a whole.

Multiple commands and pipelining

A client can use the same connection in order to issue multiple commands. Pipelining is supported so multiple commands can be sent with a single write operation by the client, without the need to read the server reply of the previous command before issuing the next one. All the replies can be read at the end.

For more information please check our page about Pipelining.

Inline Commands

Sometimes you have only `telnet` in your hands and you need to send a command to the Redis server. While the Redis protocol is simple to implement it is not ideal to use in interactive sessions, and `redis-cli` may not always be available. For this reason Redis also accepts commands in a special way that is designed for humans, and is called the **inline command** format.

The following is an example of a server/client chat using an inline command (the server chat starts with S:, the client chat with C:)

```
C: PING
S: +PONG
```

The following is another example of an inline command returning an integer:

```
C: EXISTS somekey
S: :0
```

Basically you simply write space-separated arguments in a telnet session. Since no command starts with `*` that is instead used in the unified request protocol, Redis is able to detect this condition and parse your command.

High performance parser for the Redis protocol

While the Redis protocol is very human readable and easy to implement it can be implemented with a performance similar to that of a binary protocol.

RESP uses prefixed lengths to transfer bulk data, so there is never need to scan the payload for special characters like it happens for instance with JSON, nor to quote the payload that needs to be sent to the server.

The Bulk and Multi Bulk lengths can be processed with code that performs a single operation per character while at the same time scanning for the CR character, like the following C code:

```
#include <stdio.h>

int main(void) {
    unsigned char *p = "$123\r\n";
    int len = 0;

    p++;
    while(*p != '\r') {
        len = (len*10)+(*p - '0');
        p++;
    }

    /* Now p points at '\r', and the len is in bulk_len. */
    printf("%d\n", len);
    return 0;
}
```

After the first CR is identified, it can be skipped along with the following LF without any processing. Then the bulk data can be read using a single read operation that does not inspect the payload in any way. Finally the remaining the CR and LF character are discarded without any processing.

While comparable in performance to a binary protocol the Redis protocol is significantly simpler to implement in most very high level languages, reducing the number of bugs in client software.