Duncan Department of Statistics, UC Davis Temple Lang, Department of Statistics, UC Davis

Table of Contents

Overview

The RCurl package provides high-level facilities in R to communicate with HTTP servers. Simply, it allows us to download URLs, submit forms in different ways, and generally compose HTTP requests. It supports HTTPS, the secure HTTP; handles authentication using passwords; and can use FTP to download files. It also handles escaping characters in requests, binary data, and file uploads. Users can override or provide additional headers in the HTTP request in order to customize the communication. The response from the HTTP server is processed as a stream and chunk encoding automatically handled. While the default mechanism simply returns the text from a request, one can specify S functions to process the response as it is received, redirecting it or processing in an application specific manner.

All of this could be written in R. We could use socket connections to write requests to HTTP servers and receive the response. To support HTTPS, we would have to add SSL connections. To get the same behavior as RCurl, we would have to implement the HTTP protocol. This involves writing the HTTP headers correctly and flexibly, escaping characters, providing authentication, following redirection commands, handling responses by decomposing "chunked" content, binary files, etc. To submit forms, we have to compose the body of the request by computing boundary strings and creating the "Content-Disposition" elements. In short, there is a lot of work to do and significant potential for error. Rather than doing this in R, RCurl uses an existing implementation that is provided in a widely used C library - libcurl. This has several benefits:

- libcurl is well tested and very portable. As a result it is available on many platforms and bugs have been identified and fixed.

- Many people and applications use libcurl which means that it has support for common features that are used in various contexts.

- libcurl is in C and so is fast.

On the negative side, it is hard to extend libcurl. We can only hope that the hooks to customize requests are adequate for our needs. We can expect that others have run into these extensibility issues and these needs have been fed back to the designers of libcurl.

R already provides its own way to download URIs. Functions like

`download.url`

and connection constructors such as

`url`

.

Since the libcurl library provides opaque data structures for implementing requests and connections, we cannot easily adapt it to fit our special needs in R. Specifically, it is harder to merge it with our view of connections in S (R and S-Plus). Similarly, it is harder to integrate it with the event-loop in R so that we can listen for input pending and essentially put the connection *in the background*. The support for threads in libcurl might make this easier if/when we support threads in R.

A Quick Tour

This document aims to provide a basic overview of the RCurl package. It doesn't try to provide all the details. The R function help files and the libcurl documentation have all the relevant information. Since the package is an interface to libcurl, it is important to use the documentation for it regarding features, options, etc. You can consult the libcurl documentation and libcurl examples (in C code).

The RCurl package provides three primary high-level entry points. These allow us to fetch a URL and submit forms. The functions are

`getURL`

,

`getForm`

and

`postForm`

. The first is relatively straightforward, given the name; it allows us to fetch the contents of a URI. The other two functions provide ways to submit a form using the GET or POST methods. These are quite different internally, but for users, both require a set of name-value pairs giving the parameters for the form submission. The difference is in how the form is submitted and the POST method allows us to submit/upload files, binary content, etc.

Let us look at the

`getURL`

function. At it simplest, this is just like the

`download.url`

function in the standard R. We can fetch a URI with the command something like

```
getURL("http://www.omegahat.org/RCurl/index.html")
```

The idea is that we specify the URI. There are several other arguments to this function, but for the most part we don't need them.

We can use HTTPS to fetch URIs securely. For example,

`getURL("https://sourceforge.net")`
This is already more than we can do with the regular connections or built-in

`download.url`
in R. (Using an external program allows HTTPS access.)

There are three different sets of arguments for the

`getURL`
function. One is named *`curl`* and we will cover this in section the section called “CURL Handles” [10]. This is merely a way to cumulate requests on a single connection with shared options.

The *`write`* function is again rather specialized. It allows us to specify an R function that is called each time libcurl has some text as part of the HTTP response. It hands this text (as a sequence of bytes) to the function so that it can process it in whatever way it deems fit. This corresponds to the writefunction option for the libcurl operation described next. We have it as an explicit argument simply because we need to use it to get the return value in a single action as the default behavior.

The third set of arguments is the most general and is handled by the ... in the

`getURL`
function. With this, one can specify name-value pairs governing the actual request. There are numerous possible settings that one can specify. The basic idea is that one can set options provided by the *curl_easy_setopt* routine. These allow us to set parameters for many different aspects of the request. For example, we can specify additional headers for the HTTP request, or include a password for the Web site. The set of possible options can be determined via the function

`getCurlOptionConstants`
. and the set of names for the different options can be found via the command

`names(getCurlOptionsConstants())`
This is a collection of names of options that are understood by many of the functions in the RCurl package.

At present, there are 113 possible options.

```
sort(names(getCurlOptionsConstants()))
  [1] "autoreferer"             "buffersize"
  [3] "cainfo"                  "capath"
  [5] "closepolicy"             "connecttimeout"
  [7] "cookie"                  "cookiefile"
  [9] "cookiejar"               "cookiesession"
 [11] "crlf"                    "customrequest"
 [13] "debugdata"               "debugfunction"
 [15] "dns.cache.timeout"       "dns.use.global.cache"
 [17] "egdsocket"               "encoding"
 [19] "errorbuffer"             "failonerror"
 [21] "file"                    "filetime"
 [23] "followlocation"          "forbid.reuse"
 [25] "fresh.connect"           "ftp.create.missing.dirs"
 [27] "ftp.response.timeout"    "ftp.ssl"
 [29] "ftp.use.eprt"            "ftp.use.epsv"
 [31] "ftpappend"               "ftplistonly"
 [33] "ftpport"                 "header"
```

```
 [35] "headerfunction"          "http.version"
 [37] "http200aliases"          "httpauth"
 [39] "httpget"                 "httpheader"
 [41] "httppost"                "httpproxytunnel"
 [43] "infile"                  "infilesize"
 [45] "infilesize.large"        "interface"
 [47] "ipresolve"               "krb4level"
 [49] "low.speed.limit"         "low.speed.time"
 [51] "maxconnects"             "maxfilesize"
 [53] "maxfilesize.large"       "maxredirs"
 [55] "netrc"                   "netrc.file"
 [57] "nobody"                  "noprogress"
 [59] "nosignal"                "port"
 [61] "post"                    "postfields"
 [63] "postfieldsize"           "postfieldsize.large"
 [65] "postquote"               "prequote"
 [67] "private"                 "progressdata"
 [69] "progressfunction"        "proxy"
 [71] "proxyauth"               "proxyport"
 [73] "proxytype"               "proxyuserpwd"
 [75] "put"                     "quote"
 [77] "random.file"             "range"
 [79] "readfunction"            "referer"
 [81] "resume.from"             "resume.from.large"
 [83] "share"                   "ssl.cipher.list"
 [85] "ssl.ctx.data"            "ssl.ctx.function"
 [87] "ssl.verifyhost"          "ssl.verifypeer"
 [89] "sslcert"                 "sslcertpasswd"
 [91] "sslcerttype"             "sslengine"
 [93] "sslengine.default"       "sslkey"
 [95] "sslkeypasswd"            "sslkeytype"
 [97] "sslversion"              "stderr"
 [99] "tcp.nodelay"             "telnetoptions"
[101] "timecondition"           "timeout"
[103] "timevalue"               "transfertext"
[105] "unrestricted.auth"       "upload"
[107] "url"                     "useragent"
[109] "userpwd"                 "verbose"
[111] "writefunction"           "writeheader"
[113] "writeinfo"
```

Each of these and what it controls is described in the libcurl man(ual) page for *curl_easy_setopt* and that is the authoritative documentation. Anything we provide here is merely repetition or additional explanation.

The names of the options require a slight explanation. These correspond to symbolic names in the C code of libcurl. For example, the option url in R corresponds to
<c:enumValue>CURLOPT_URL</c:enumValue>
in C. Firstly, uppercase letters are annoying to type and read, so we have mapped them to lower case letters in R. We have also removed the prefix "CURLOPT_" since we know the context in which they option names are being used. And lastly, any option names that have a _ (after we have removed the CURLOPT_

prefix) are changed to replace the '_' with a '.' so we can type them in R without having to quote them. For example, combining these three rules, "CURLOPT_URL" becomes url and
<c:enumValue>CURLOPT_NETRC_FILE</c:enumValue>
becomes netrc.file. That is the mapping scheme.

The code that handles options in RCurl automatically maps the user's inputs to lower case. This means that you can use any mixture of upper-case that makes your code more readable to you and others. For example, we might write `writeFunction = basicTextGatherer()` or `HTTPHeader = c(Accept="text/html")`

We specify one or more options by using the names. To make interactive use easier, we perform partial matching on the names relative to the set of know names. So, for example, we could specify

```
getURL("http://www.omegahat.org/RCurl/testPassword",
       verbose = TRUE)
```

or, more succinctly,

```
getURL("http://www.omegahat.org/RCurl/testPassword",
       v = TRUE)
```

Obviously, the first is more readable and less ambiguous. Please use the full form when writing "software". But you might use the abbreviated form when working interactively.

Each option expects a certain type of value from R. For example, the following options expect a number or logical value.

```
[1] "autoreferer"             "buffersize"
 [3] "closepolicy"             "connecttimeout"
 [5] "cookiesession"           "crlf"
 [7] "dns.cache.timeout"       "dns.use.global.cache"
 [9] "failonerror"             "followlocation"
[11] "forbid.reuse"            "fresh.connect"
[13] "ftp.create.missing.dirs" "ftp.response.timeout"
[15] "ftp.ssl"                 "ftp.use.eprt"
[17] "ftp.use.epsv"            "ftpappend"
[19] "ftplistonly"             "header"
[21] "http.version"            "httpauth"
[23] "httpget"                 "httpproxytunnel"
[25] "infilesize"              "ipresolve"
[27] "low.speed.limit"         "low.speed.time"
[29] "maxconnects"             "maxfilesize"
[31] "maxredirs"               "netrc"
[33] "nobody"                  "noprogress"
[35] "nosignal"                "port"
[37] "post"                    "postfieldsize"
[39] "proxyauth"               "proxyport"
[41] "proxytype"               "put"
[43] "resume.from"             "ssl.verifyhost"
[45] "ssl.verifypeer"          "sslengine.default"
[47] "sslversion"              "tcp.nodelay"
[49] "timecondition"           "timeout"
[51] "timevalue"               "transfertext"
```

```
[53] "unrestricted.auth"       "upload"
[55] "verbose"
```

The connecttimeout gives the maximum number of seconds the connection should take before raising an error, so this is a number. The header option, on the other hand, is merely a flag to indicate whether header information from the response should be included. So this can be a logical value (or a number that is 0 to say FALSE or non-zero for TRUE.) At present, all numbers passed from R are converted to *long* when used in libcurl.

Many options are specified as strings. For example, we can specify the user password for a URI as

```
getURL("http://www.omegahat.org/RCurl/testPassword/index.html", userpwd = "bob:dun
```

Note that we also turned on the "verbose" option so that we can see what libcurl is doing. This is extremely convenient when trying to understand why things aren't working (or are working in a particular way!).

Another example of using strings is to specify a referer URI and a user-agent.

```
getURL("http://www.omegahat.org/RCurl/index.html", useragent="RCurl", referer="htt
```

(Again, you might want to turn on the "verbose" option to see what libcurl is doing with this information.)

The libcurl facilities allow us to not only set our own values for fields used in the HTTP request header (such as the referer or user-agent), but it also allows us to set an entire collection of new fields or replacements for any existing field. We do this in R using the httpheader option for libcurl and we specify a value which is a named character vector. For example, suppose we want to provide a value for the Accept field and add a new field named, say, Made-up-field. We could do this in the request as

```
getURL("http://www.omegahat.org/RCurl", httpheader = c(Accept="text/html", 'Made-u
```

If you turn on the verbose option again for this request, you will see these fields being set.

```
> getURL("http://www.omegahat.org", httpheader = c(Accept="text/html", 'Made-up-fi
* About to connect() to www.omegahat.org port 80
* Connected to www.omegahat.org (169.237.46.32) port 80
> GET / HTTP/1.1
Host: www.omegahat.org
Pragma: no-cache
Accept: text/html
Made-up-field: bob
```

(Note that not all servers will tolerate setting header fields arbitrarily and may return an error.)

The key thing to note is that headers are specified as name-value pairs in a character vector. R takes these and pastes the name and value together and passes the resulting character vector to libcurl. So while it is convenient to express the headers as

```
c(name = "value", name = "value")
```

if you already have the data in the form

```
c("name: value", "name: value")
```

you can use that directly.

Some of the libcurl options expect a C routine. For example, when libcurl is receiving the response from the HTTP server, it will call the C routine specified via the option
<c:enumValue>CURLOPT_WRITEFUNCTION</c:enumValue>

each time it has a full buffer of bytes. While it is possible for us to be able to specify a C routine from R (using

`getNativeSymbolInfo`

), we currently don't support this. Instead, it is more natural to specify an R function which is to be called when appropriate. And this is indeed how we do things in RCurl. One can specify a function for the writefunctionwriteheader and debugfunction options. (We can add support for the others such as readfunction.) To use these is quite simple. We expect an R function that takes a single argument which is the character of bytes to process. The function can do what it wants with this argument. Typically, it will accumulate it in a persistent variable (e.g. using closures) or process it on-the-fly such as adding to a plot, passing it to an HTML parser, ...

The function

`basicTextGatherer`

is an example of the idea and this mechanism is used in

`getURL`

. Suppose, for some reason, we wanted to read the header information that was returned by HTTP server in the response to our request. (This has interesting things like cookies, content type, etc. that libcurl uses internally, but we may also want to process.) Then we would firstly use the header option to turn on the libcurl facility to report the response header information. If we just do this, the header information will be included in the text that

`getURL`

returns. This is fine, but we will have to separate it out by finding the first line, etc. Instead, it is easier to ask libcurl to hand the header information to use separate from the text/body of the response. We can do this by creating a callback function via the

`basicTextGatherer`

function.

```
h = basicTextGatherer()
 txt = getURL("http://www.omegahat.org/RCurl", header = TRUE, headerfunction = h$u
```

All we have done is create a collection of functions (stored in **h**) and passed the update callback to libcurl. Each time libcurl receives more of the headers, it calls this function with the header text. It may call this just once or several times. This depends on how large the header information is, how libcurl buffers the information, etc.

Having called

`getURL`

, we have the text from the URI. The header information is available from **h**, specifically its **`value`** function element.

```
h$value()
```

The

`debugGatherer`

is another example of a callback that can be used with libcurl. If we set the "verbose" option to **`TRUE`**, libcurl will provide a lot of information about its actions. By default, these will be written on the console (e.g. stderr). In some cases, we would not want these to be on the screen but instead, for example, displayed in a GUI or stored in a variable for closer examination. We can do this by providing a callback function for the debugging output via the debugfunction option for libcurl. The

```
debugGatherer
```
is a simple one that merely cumulates its inputs in different categories and makes them available via the **value** function. The setup is easy:

```
d = debugGatherer()
  x = getURL("http://www.omegahat.org/RCurl", debugfunction=d$update, verbose = TR
```
At the end of the request, again we have the text from the URI in **x**, but we also have the debugging information. libcurl has called our **update** function each time it has some information (either from the HTTP server or from its own internal dialog).

```
(R) names(d$value())
[1] "text"      "headerIn"  "headerOut" "dataIn"    "dataOut"
```
The headerIn and headerOut fields report the text of the header for the response from the Web server and for our request respectively. Similarly, the dataIn and dataOut fields give the body of the response and request. And the text is just messages from libcurl.

We should note that not all options are (currently)) meaningful in R. For example, it is not *currently* possible to redirect standard error for libcurl to a different *FILE** via the "stderr" option. (In the future, we may be able to specify an R function for writing errors from libcurl, but we have not put that in yet.)

Forms

The RCurl package provides many additional mechanisms for downloading URIs that R does not currently have built-in. But perhaps the most pressing reason for developing the RCurl package was the need to submit forms. The [1] [18] package is a package that can read an HTML page with one or more forms and create an S function for each form that allows S users to submit the form programmatically rather than requiring interactively browsing the page, saving the result to a file and then loading it into R. In order for these functions to work, we need to be able to submit the contents of the form from S as if it came from a regular browser. We use RCurl to do this.

There are two mechanisms used for submitting HTML forms: GET and POST. Both take a set of name-value pairs giving the arguments to parameterize the call. The difference between the mechanisms is how these name-value pairs are delivered to the HTTP server. The GET method puts the name-value pairs of parameters at the end of the URI name, e.g.

http://www.omegahat.org/cgi-bin/form.pl?a=1&b=2

The POST method expects the name-value pairs to be sent as the body of the HTTP request, each put in its own "paragraph" or stanza. This is more complicated but supports sending binary data, etc.

Which of the GET and POST mechanism is appropriate is specified with the HTML form itself via the `action` attribute of the `<FORM>` itself. To the user, however, the browser takes care of figuring out the correct way to deliver the name-value pairs specified by the user when interacting with the components of the form. In RCurl, we don't have access to the original HTML form so we cannot tell what mechanism to use. It is up to the caller to determine whether to use

```
getForm
```
or

`postForm`
depending on the value of the `action` attribute in the original HTML file.

After determining whether to use POST or GET, the interface to the functions is typically the same to the user. Essentially, she need only specify the name-value pairs for each of the form elements. We do this via a named list or named character vector. (The list simply allows us to have objects of different type other than strings!) We must specify all the fields, including the hidden fields, if the the processor on the HTTP server is to make sense of it. RCurl doesn't try to interpret the name-value pairs, but just transports them.

Let's look at an example of sending a query to Google (via HTTP rather than its API).

```
getForm("http://www.google.com/search", hl="en", lr="", ie="ISO-8859-1",  q="RCurl
```

The result is the HTML you would ordinarily see in your browser. You might use

`htmlTreeParse`
to parse it. What is important in the example is that we are specifying the required fields in the query as named arguments to R.

`getForm`
takes care of bringing them together and constructing the full URI name. Note that libcurl also handles escaping the special characters, e.g. converting a space to %20. Note that if you wanted to explicitly do this escaping on a string rather than having libcurl implicitly do it, you can use

`curlEscape`
. Similarly, there is a function

`curlUnescape`
to reverse the escaping and make a string "human-readable".

`postForm`
is almost identical. Let's submit a POST form to http://www.speakeasy.org/~cgires/perl_form.cgi

```
postForm("http://www.speakeasy.org/~cgires/perl_form.cgi",
            "some_text" = "Duncan",
            "choice" = "Ho",
            "radbut" = "eep",
            "box" = "box1, box2"
          )
```

Here, the form elements are named some_text, choice, radbut, box. We have simply provided values for them. Again, the result is the regular response from the HTTP server.

Sometimes we already have the arguments in a list. It is slightly more complex then to pass them to the function via the **...** argument. The two form submission functions in RCurl (

`getForm`
and

`postForm`
) also accept the name-value arguments via the **...** parameter. This arises in programmatic access to the functions rather than interactive use.

Since we use **...** for the name-value pairs of the form, we cannot specify the libcurl options (unambiguously) in this way and we require than any such options to control the HTTP request at the libcurl-level be passed

via the *.opts* parameter. RCurl and libcurl construct the HTTP request and after that, the request is just like a regular URI download. All of the usual techniques for reading the response, its header, etc. work.

CURL Handles

The functions we have presented above are the high-level entry points that allow R users to make the common-style HTTP requests. The RCurl package is capable of more however. It provides access to the basic libcurl primitives which one can use to compose more complicated and non-standard HTTP requests. For the most part, one merely specifies libcurl options by name to the different functions and these take effect for that call. An alternative model (used more in C code) is that we first create a libcurl object to represent the HTTP request, then we customize it by setting options and then we invoke the request. This is far more involved than we need in R. There is a simplicity about the

`getURL`

function that removes the need to know about the internal C structure representing the call. However, there are occasions when it is useful to know about this and exploit it. Specifically, one can create an instance of this libcurl "handle" and use it in several requests. This has the advantage that we do not have to set the options in each call, but rather can do this just once. This saves a marginal amount of time in R by reducing the computations, but it will be essentially negligible relative to the network latency involved in the request itself. What is more important is that if the sequence of requests are to the same server, the libcurl engine can maintain the connection to the server and avoid having to reestablish it each time. This handshaking is quite expensive, so reusing the "handle" in such situations can yield non-trivial performance gains. It is also even possible to "pipeline" requests by sending multiple requests before getting the answer back for the first one. This again can improve performance.

Now that we both know about the internal libcurl structures and know why we might be interested in reusing them across requests, the question remains how do we do this. It is quite easy. Each of the "action" functions in the package (i.e. that work with libcurl directly) have a parameter named *curl*. For each of these functions, the default value is

`getCurlHandle`

and what this means is that, if no value is given for *curl*, a new handle is created for the duration of this call. So it is easy for us to create such a handle before calling one of these functions and then pass that as the value for *curl*. For example, we can make two requests to the www.omegahat.org site using the same handle as follows:

```
handle = getCurlHandle()
 a = getURL("http://www.omegahat.org/RCurl", curl = handle)
 b = getURL("http://www.omegahat.org/", curl = handle)
```

It is important to remember that if we set any options in any of the calls, these will be set in the libcurl handle and these will persist across requests unless they are reset. For example, if we had set the `header=TRUE` option in the first call above, it would remain set for the second call. This can be sometimes inconvenient. In such cases, either use separate libcurl handles, or reset the options.

The function

`dupCurlHandle`

allows us to create a new libcurl handle that is an exact copy of the existing one. This allows us to quickly reuse existing settings without having them affect other requests. (The data in the option values are not copied). See *curl_easy_duphandle*.

By reusing libcurl handles, we avoid reallocating a new one and potentially benefit from improved connectivity. One downside, however, when reusing handles is that the options we set in R need to be copied as C data since they will persist across R function calls in the libcurl handle itself. As a result, there are additional computations needed. Again, this is negligible in almost all cases and will be dominated by the network speed.

libcurl doesn't have any explicit function for fetching a URL. Instead, it uses a powerful but simple interface which involves merely setting the options in the libcurl handle as desired and then invoking the request. So one just prepares the request and forces it to be sent. This is done via the

```
curlPerform
```

function in R. This is how

```
getURL
```

is actually implemented.

Statistics on the Request

S is a statistical programming language and environment so why not gather data when we can. The function

```
getCurlInfo
```

allows us to find out information about the last request made for a given libcurl handle. (Of course, this assumes we have explicitly created the handle rather than used the default value for `curl` and so lost it.)

```
h = getCurlHandle()
getURL("http://www.omegahat.org", curl = h)
names(getCurlInfo(h))
```

The names of the resulting elements are

```
[1] "effective.url"           "response.code"
 [3] "total.time"              "namelookup.time"
 [5] "connect.time"            "pretransfer.time"
 [7] "size.upload"             "size.download"
 [9] "speed.download"          "speed.upload"
[11] "header.size"             "request.size"
[13] "ssl.verifyresult"        "filetime"
[15] "content.length.download" "content.length.upload"
[17] "starttransfer.time"      "content.type"
[19] "redirect.time"           "redirect.count"
[21] "private"                 "http.connectcode"
[23] "httpauth.avail"          "proxyauth.avail"
```

These provide us the actual name of the URI downloaded after redirections, etc.; information about the transfer speed, etc.; etc. See the man page for *curl_easy_getinfo*.

libcurl Version Information

The RCurl package provides a way to obtain reflectance information about libcurl itself. The function

```
curlVersion
```

returns the contents of the
<c:struct>curl_version_info_data</c:struct>
structure.

For my installation, the return value from

```
curlVersion
```

is

```
$age
[1] 2

$version
[1] "7.12.0"

$vesion_num
[1] 461824

$host
[1] "powerpc-apple-darwin7.4.0"

$features
     ipv6       ssl      libz      ntlm largefile
        1         4         8        16       512

$ssl_version
[1] " OpenSSL/0.9.7b"

$ssl_version_num
[1] 9465903

$libz_version
[1] "1.2.1"

$protocols
[1] "ftp"    "gopher" "telnet" "dict"   "ldap"   "http"   "file"   "https"
[9] "ftps"

$ares
[1] ""

$ares_num
[1] 0

$libidn
[1] ""
```

The help page for the R function explains the fields which are hopefully clear from the names. The only ones that might be obscure are **`ares`** and **`libidn`**. **`ares`** refers to asynchronous domain name server (DNS) lookup for resolving the IP address (e.g. 128.41.12.2) corresponding to a machine name (e.g. www.omegahat.org). "GNU Libidn is an implementation of the Stringprep, Punycode and IDNA specifications defined by the IETF Internationalized Domain Names (IDN)" (taken from http://www.gnu.org/software/libidn/).

Initialization

As with most C libraries, one must typically initialize it to get the basic run-time structure established. Fortunately, when we call any of the R functions, this is taken care of implicitly; libcurl handles this when we create a new handle. However, sometimes it is important to explicitly specify options to control how the library is initialized. RCurl provides a way to do this via the

`curlGlobalInit`

function. The only argument is a flag that indicates what features to initialize. For the most part, the defaults work best and we can leave libcurl to perform this initialization. However, we may need to be careful that we are not re-initializing a setting that R (or generally the host application) has already set. This may happen on Windows as libcurl initialize the Win32 socket library. We can avoid this, if necessary, but telling libcurl to initialize only the SSL facilities.

The argument to

`curlGlobalInit`

is typically a character vector of names of features to turn on. The possible names can be obtained from **`CurlGlobalBits`** which is a named integer vector:

```
none   ssl win32   all
    0     1     2     3
attr(,"class")
[1] "CurlGlobalBits" "BitIndicator"
```

We would call

`curlGlobalInit`

as

`curlGlobalInit(c("ssl", "win32"))`

or

`curlGlobalInit(c("ssl"))`

to activate both SSL and Win32 sockets, or just SSL respectively.

One can specify integer values directly, but this is less readable to others (or yourself in a few weeks!). The names are converted and combined to a flag using

`setBitIndicators`

.

Options

It is hopefully clear that it is the libcurl options that make this interface work and allow us to make interesting queries. From specifying the URI to how to read the text, to providing passwords, it is the options that are critical. For the most part, these options are passed by name to functions in RCurl via the **...** mechanism in R and the *`.opts`* argument. These two collections of arguments are merged, with those in **...** overriding corresponding ones in the *`.opts`* object.

Why do we have the *`.opts`* argument? The reason is similar to the *`.params`* in the form functions: often we have the options in a list and it is not as convenient to use the **...** approach. Having both allows the caller/programmer to use whichever is most convenient.

One case in which the `.opts` argument is useful is if we want to prepare a set of options that are to be used in all (or a set of) calls. We can combine these arguments into a list just once and then pass them to each HTTP request easily by simply using that variable. Since we merge the values in **...** and `.opts`, this works nicely.

To create such a list of options, we can use the function

`curlOpts`

. This creates an S3-style object with class
<s:class>CURLOptions</s:class>
. This function never involves libcurl, but sorts out the names of the options by using partial matching (via the

`mapCurlOptNames`

function) and returns an R object with the options as name-value pairs in a list. The fact that this is a class means that if we access any elements, the full names are used, even when we set an element. This means that the names are kept resolved as we use it in R and correspond unambiguously to real libcurl options.

We can use this function something like the following.

```
opts = curlOptions(header = TRUE, userpwd = "bob:duncantl", netrc = TRUE)
getURL("http://www.omegahat.org/RCurl/testPassword/index.html", verbose = TRUE, .o
```

Here we create the options ahead of time and use them in a call while specifying additional options (i.e. "verbose").

Some readers will have noticed that we could achieve the same effect of having a set of fixed options that are used in a collection of calls by reusing a libcurl handle. We could create the handle, set the common options, and then use that handle in the set of calls. This is indeed a natural and often good way to do things. The following code does what we want.

```
h = getCurlHandle(header = TRUE, userpwd = "bob:duncantl", netrc = TRUE)
getURL("http://www.omegahat.org/RCurl/testPassword/index.html", verbose = TRUE, cu
```

The first line creates a new handle and fills in the three "persistent" options. These are in the handle itself, not in R at this stage. Now, when we perform the request via

`getURL`

, we specify this libcurl handle and provide the "verbose" option.

The function

`curlSetOpt`

is used implicitly in the code above and this actually sets the option-values in a libcurl handle. It can also be used to simply resolve them.

Example Application

In this section, we will outline a more complex use of the RCurl facilities. Specifically, we will use it to send SOAP [2] [18] requests. SOAP uses HTTP to send XML content that encodes a method invocation. We have to add the appropriate fields to the HTTP header to identify the SOAP call and then insert the XML that defines the method call in the body of the request. Using our favorite client SOAP facility (e.g. SOAP::Lite, SSOAP), we can send the SOAP request that performs the HTTP request. We can find out what the actual HTTP request looks like using facilities in those tools or actually sniffing the packets as they go across the wire using tcpdump or ethereal or some such tool. This is what we see. The HTTP header in the request is

POST /hibye.cgi HTTP/1.1
Connection: close
Accept: text/xml
Accept: multipart/*
Host: services.soaplite.com
User-Agent: SOAP::Lite/Perl/0.55
Content-Length: 450
Content-Type: text/xml; charset=utf-8
SOAPAction: "http://www.soaplite.com/Demo#hi"

The body of the request is

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
              xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
              xmlns:xsd="http://www.w3.org/1999/XMLSchema"
              xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
              xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
  <SOAP-ENV:Body>
     <namesp1:hi xmlns:namesp1="http://www.soaplite.com/Demo"/>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

So we need to add fields to the HTTP header. Specifically, we need

```
Accept: text/xml
Accept: multipart/*
SOAPAction: "http://www.soaplite.com/Demo#hi"
Content-Type: text/xml; charset=utf-8
```

libcurl should take care of the Content-Length field. The body is specified for the HTTP request using the postfields option. To do this using the *RCurl* package, we use the following code.

```
body = '<?xml version="1.0" encoding="UTF-8"?>\
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encodin
                   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" \
                   xmlns:xsd="http://www.w3.org/1999/XMLSchema" \
                   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/" \
                   xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">\
  <SOAP-ENV:Body>\
       <namesp1:hi xmlns:namesp1="http://www.soaplite.com/Demo"/>\
  </SOAP-ENV:Body>\
</SOAP-ENV:Envelope>\n'

curlPerform(url="http://services.soaplite.com/hibye.cgi",
```

```
            httpheader=c(Accept="text/xml", Accept="multipart/*", SOAPAction='"htt
                         'Content-Type' = "text/xml; charset=utf-8"),
            postfields=body,
            verbose = TRUE
           )
```

Note that this similar to calling

`getURL`

and we have used it to illustrate how we can use

`curlPerform`

directly. The only difference is that the result is printed to the console, not returned to us as a character vector. This is a problem when we really want to process the response. So for that, we would simply replace the call to

`curlPerform`

with

`getURL`

:

```
curlPerform(url="http://services.soaplite.com/hibye.cgi",
            httpheader=c(Accept="text/xml", Accept="multipart/*", SOAPAction='"htt
                         'Content-Type' = "text/xml; charset=utf-8"),
            postfields=body,
            verbose = TRUE
           )
```

Additional Notes

These were written before the package was finished. They are left here, but may not be helpful.

Providing Text Handlers

One can provide a different text handler to consume/process the text that is received by the libcurl engine from the HTTP response. We can do this with either an S function or with a C routine. At the lowest level, there is a C routine, but R users will typically be most comfortable with a higher-level function. Functions are more robust, and also provide a more obvious way of maintaining state. We use closures and environments in R to endow a function with its own local variables that persist across calls.

One can envisage a scenario in which the text handler would want to be able to access the CURL handle that was being used in the request. For example, it might use this to find the base URL, to determine the actual host, or identify settings that are in effect, or simply reuse the handle (by duplicating it). Rather than make the curl object explicitly available, one can initialize it separately ahead of the call to

`getURL`

, and then make it available to the R function as a variable in the function's environment. A more realistic example is the following. Suppose we want to parse HTML files and follow the links within those files to find their links. This is a spider or 'bot. We can fetch the entire document via

`getURL`

and then parse it (using

`htmlTreeParse`

). Alternatively, we can use

`htmlTreeParse`

and provide it with a connection from which the HTML/XML parser requests content as it is needed. We can then setup this connection from which the XML parser reads by having it be supplied by the

`getURL`

text gatherer. We can parse a document and collect the names of all the links to which it refers and then process each of them. An alternative is to process an individual link when it is discovered. There are trade-offs between the two approaches. However, it is good to be able to do both.

In order to do this, we might want to have access to the CURL handle within the XML parser. When we handle an HREF element (i.e. <a href=>), we would then duplicate the handle and start another HTML parser.

Obviously, this example is also somewhat contrived as the XML/HTML parsing facilities have their own HTTP facilities. However, they do not understand all the finer points of HTTP such as SSL, FTP, passwords, etc.

Note

This is not a very compelling example anymore!

Alternatives

Using libcurl is by no means the only approach to getting HTTP access in R. Firstly, we have HTTP access in R via the facilities incorporated from libxml (nanohttp and nanoftp). These are, as the names suggest, basic implementations of the protocols and do not provide all the bells and whistles we might need generally. Also, they are not customizable from within R. Specifically, we cannot add header fields, handle binary data, set the body of the request, etc.

We can use R's socket connections and implement the details of HTTP ourselves. There is a great deal of work in this as we have discussed before. Also, we currently don't have secure sockets (i.e. using SSL) in R[1] I initially started using this approach so that I could discover the nuances of HTTP. It quickly gets overwhelming to handle all the details. It is more tedious than technically challenging, especially when others have done it already in C libraries and done it well. The code that I have is in an unreleased package named *`httpClient`*. If anyone is interested, please contact me. Using R's sockets is also used in the httpRequest package on CRAN. This allows submitting forms and retrieving URIs. It is useful and, as the authors state, a "basic HTTP request" implementation. It doesn't escape characters, handle chunked responses, do redirects, support SSL, etc. It is flexible but leaves a lot to the user to do to setup the request and process the response. RCurl inherits many, many good features for "free" from libcurl.

libcurl is not the only C-level library that we could have used. Alternative libraries include libwww from the W3 group. We *may* find that that is more suitable, but libcurl will definitely suffice for the present.

Notes

libcurl can use ares for asynchronous DNS resolution.

[1] I have a local version (not with SSL) but they are not connections since the connection data structure is not exposed in the R API, yet!

Bibliography

[1] Sandrine Dudoit, Sunduz Keles, and Duncan Temple Lang. *The odbAccess package: creating S functions from HTML forms.*. odbAccess (coming soon)

[2] James Snell, Doug Tidwell, and Pavel Kulchenko. *Programming Web Services with SOAP*. O'Reilly