Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *http://www.jstatsoft.org/*

R as a Web Client – the RCurl package

Duncan Temple Lang
Department of Statistics,
University of California at Davis

Abstract

The Web is clearly an important source of data for statisticians as is emerging as vital component in distributed computing via Web services. HTTP is the primary mechanism that underlies the Web and data transfer. As such, it is important for programming languages to have tools for HTTP requests and other protocols. We describe the **RCurl** package that provides an interface to a general HTTP and Web protocol library – libcurl. This paper provides an overview of the user-level facilities provided to R programmers by this interface, as well as some simple examples as to how to use these. More detailed examples of advanced topics are also discussed, focusing on asynchronous requests. Finally, we discuss alternative approaches to implementing such facilities in R. The package is available from the Omegahat Web site at `http://www.omegahat.org/RCurl`.

Keywords: HTTP, Web services, HTML, XML, data.

1. Motivation

"Web Services" is a relatively recently introduced "buzz" phrase. And, of course, the Web and the HyperText Transfer Protocol - HTTP - that underlies the communication of data on the Web have become a vital part of our information network and day to day environment. Measuring the proportion of data sent via HTTP can be difficult, but estimates range from 31% to 75% Claffy and Miller (1998), Spr (2004). Accordingly, being able to access various forms of data using HTTP is an important facility in a general programming language. We want to be able to download files, get data via HTML (HyperText Markup Language (2006) (W3C)) forms, "scrape" HTML page content as data itself, and use SOAP (Simple Object Access Protocol) Snell, Tidwell, and Kulchenko (2002) to invoke methods provided via Web Services. And, importantly, we want to be able to rapidly take advantage of new and future technologies as they emerge that leverage the increasingly ubiquitous HTTP and other Web protocols.

The connections framework of S4 Chambers (1999) provided a general model for input and

output streams for the S language. The implementation of this, primarily by Brian D. Ripley, for the R environment R Development Core Team (2005) has incorporated the HTTP (and FTP) facilities of the libxml suite Veillard (2006) and allowed R to provide network facilities. These provide R with important tools for downloading documents via HTTP. However, they do not allow the user to control various essential aspects for *general* HTTP use. They only support, in the words of Daniel Veillard, the author of the libxml tools,

> *a minimalist HTTP GET ... This is clearly not a general purpose HTTP implementation.*

It does not support the POST mechanism in HTTP for submitting forms, or uploading files. Importantly, it does not support secure HTTP (HTTPS) – HTTP over SSL (Secure Socket Layer) Viega, Messier, and Chandra (2002) which provide secure, confidential, encrypted connections. Additionally, R users cannot provide information that governs the connection to the HTTP server. Information in the header part of an HTTP request such as persistent connections, the identity of the user agent and passwords for authentication are not specifiable via this interface. At the content-level for a request, one has to provide the HTTP request in explicit form, converting special characters to their HTTP equivalent. For example, literal spaces in the document name must be specified as %20.

The existing support in R for HTTP facilities has been terrifically valuable. However, these are intended only for existing, basic use and are not extensible at the user programming level. They cannot therefore be easily used as-is for accessing Web services or any customized Web applications in R. Since the code was copied from libxml into the R code base (rather than dynamically linked), any enhancements to the basic engine in libxml will not be accessible to R without repeating the work. And, importantly, this means security concerns that have manifested themselves in libxml are also potential vulnerabilities in R.

Given the increasing role of HTTP and Web connectivity, and the desire to use R in ways that can access data and services in other domains via HTTP, a more general, flexible and complete R-language interface to client-side HTTP is desirable. And ideally we want to base this on evolving and maintained software whose updates can be readily incorporated into this interface with little additional work. The **RCurl** package provides such an interface for R. It is based on the cURL C-level library that underlies the commonly used `curl` command-line utility downloading documents and Web sites. It has support for numerous protocols including HTTP/HTTPS, FTP/FTPS/TFTP (trivial file transfer protocol), LDAP (Lightweight Directory Access Protocol) and is highly configurable both in the options it supports and also in how input and output can be customized by the host application, i.e. R in our case. This package provides a rich infrastructure on which the R community can build Web-based client software. It is not intended to be a direct replacement for the existing connections tools and functionality. Tools such as **download.url()** are either built into the R code base and require no additional software to be installed, or invoke external applications that are typically available on a given platform. Rather, the **RCurl** package provides a tightly integrated interface to a high quality, widely used and high-level library for Web connectivity and requires the installation of this highly portable **libcurl** library.

The remainder of this paper is organized as follows. In section 2, we provide a brief guide to HTTP and its different elements. We then discuss the RCurl package and give an overview of its facilities and the primary functions it provides in section 3. Section 4 provides some relatively simple examples of how to use the package and then we follow this with more

more advanced case studies in section 5. And finally, we discuss other approaches and some potential additions. Rather than providing terse definitions for some technical terms that are merely mentioned in the paper, descriptions of some of these terms are provided in a glossary. We do not recreate in this paper the examples in the help pages or the simple ones available on the Web site. Rather, we try to provide a more conceptual view of the package so that people can think about how to utilize it rather than merely copy existing code.

2. Overview of HTTP

In this section, we will provide a high-level, relatively brief description of various aspects of HTTP - the HyperText Transfer Protocol. It is not intended to be exhaustive, but to illustrate some of its features and show that a full implementation can quickly become complex. Those familiar with HTTP may want to skip to the end of this section.

HTTP is, in many respects, a simple mechanism to request a document from a Web server. All of us are familiar with using our Web browser to request a document At its simplest, we send the full name of the document to a Web server, e.g. `http://wwww.omegahat.org/RCurl/index.html`. This identifies

- the protocol for the request - http,
- the name of the Web server or host - www.omegahat.org
- and the fully qualified path name of the specific document (RCurl/index.html) relative the the server's top-level node.

Our client software (e.g. the browser) uses this information to communicate with the Web server. It establishes a connection to the server via a socket, typically connecting to the Web server machine's port 80. Having established the basic communication channel, the client makes the request by sending (at least) the following 3 lines:

```
1  GET /RCurl/index.html HTTP/1.1
2  Host: www.omegahat.org
3
```

These lines are quite easy to understand. The first word – `GET` – identifies the nature of the request, or the action. This means that we want to retrieve a document from the Web server. We can also `POST` information to a Web server such as uploading a file or submitting data for an HTML form. And HTTP provides several other actions: PROPERTY, OPTIONS, HEAD, PUT, DELETE, TRACE, CONNECT. The next word in the request is the name of the document being requested - /RCurl/index.html. And lastly on the first line is HTTP/1.1 which identifies the dialect of the protocol we are speaking. This tells the Web server that we are using HTTP and, in particular, version 1.1. There are two options for this - 1.0 and 1.1 and as you might guess, 1.1 is more recent and more flexible.

The second line (Host: wwww.omegahat.org) seems redundant as this is the name of the host to which we connected. However, it is necessary as it helps a Web server which hosts multiple virtual sites via the same machine. This tells the server application the identity of the virtual host so that it can process the request relative to that collection of files rather than its default collection of documents.

The third line is blank and that signifies the end of the header information for the HTTP request. Each request (and response) consists of a header and an optional body. The header information contains the request action as the first line and then a sequence of name: value pairs that parametrize the request. Each line is terminated with a control-linefeed combination i.e. the characters `\r\n`.

Given the complete header in the example above, the server can then process the details and will return its result as a collection of bytes. The response, like the request, starts with a header and is followed by the body of the response. In our example, the header looks like

```
1   HTTP/1.1 200 OK
    Date: Sun, 13 Mar 2005 15:38:26 GMT
    Server: Apache/2.0.52 (Unix)
    Last-Modified: Thu, 13 Jan 2005 18:39:10 GMT
    ETag: "e6b33-f43-3bd62f80"
    Accept-Ranges: bytes
    Content-Length: 3907
    Content-Type: text/html; charset=ISO-8859-1
9
```

Again, the header is terminated by the presence of a blank line (numbered 9). It is also made up of a first line describing the dialect of HTTP and its status (200 OK). The *name: value* pairs provide information from the server to the client about the response. They contain information about the body of the response which is the content of the requested document and how to process it. From the *Content-type* field, we know the body is text containing HTML. We know the encoding character set (ISO-8859-1), and we know how many bytes there are (3907). We also have information about the Web server and what software it is running. There are many possible fields in the request and response headers and this gives HTTP its flexibility. The interested reader is referred to Fielding, Gettys, Mogul, Frystyk, Masinter, and P. Leach (1999). More important for this paper than the details of the different parameters is the knowledge that HTTP supports a rich set of controls, and applications may need to provide and interpret these in different ways. In this respect, a flexible interface that obviates the need to know the details, but that still provides access to them is important for general use in many different contexts.

Having processed the response's header, the client software can then read the body and do what it wants with that data. It may display it in a browser, read it to extract links to other documents, display it as an image, or call additional software to process the data. The body of the response may come as a single sequence of bytes, or may be "chunked" and provided in segments. In this case, each segment identifies the number of bytes it contains and the client software is responsible for handling it appropriately and then looking for the next chunk. This is a tedious process that requires additional code to recognize the presence of chunks in the response (encoded in the header) and then to identify each successive chunk and combine the result into a single block of data for general processing. We want this to be transparent to the R programmer and end-user, but still allow advanced applications to take advantage of this when efficiency is an issue.

Retrieving documents is relatively easy with this protocol. As we have seen, the only elements that change are the host and file name. In addition to existing documents, we can also use HTTP to request dynamic or conditional content. HTML forms allow us to use our Web

browser to specify user-level inputs to an HTTP query. We select items from a menu, click on checkboxes and radio buttons and then submit our request via a button. In the simplest style of form, the browser sends the HTTP query in exactly the same way as a regular request for a document. It uses the GET action and specifies the name of the script associated with processing the form as the target document. It includes the user-specific information from the form by appending it onto the file name. Instead of /RCurl/index.html, it would include *name=value* pairs from the form in the URI (Uniform Resource Identifier) being requested. These name=value pairs are separated from each other by the '&' symbol, and separated from the file name by a '?'. For example, to send a request to a script file named /apps/myForm with two variables named first and last, the browser would construct the query

```
GET /apps/myForm?first=Duncan&last=Temple+Lang
```

Note that the space in the value of the last field (Temple Lang) is "escaped". Spaces are converted to the character '+', and non-alpha-numeric characters are represented by their hexadecimal position in the character set. Additionally, the client should indicate to the server that this HTTP request is for a form by adding

```
Content-Encoding: application/x-www-form-urlencoded
```

to the request's header.

2.1. POST and Data in the HTTP Request Body

The above method for sending user-specific information in the request is limited in several ways. If we want to upload the contents of a file in the request, the query string for the document name could be very long. Additionally, binary data would require special handling. And files that contained '?' or '&' symbols would completely confuse the Web server. These would have to be escaped in some way to ensure the server did not recognize these characters as parameter separators. Since this is a clumsy way to send arbitrary data, the designers of the HTTP specification provided a better alternative. Instead of using the GET action, HTTP provides the POST action which allows information in the request to be sent as part of the body, and not in the header. In this way, just as the response we saw earlier when requesting a document contains the data in the body as described in the header, the client software can use a header and body to transmit arbitrary, more complex data than we have seen so far to the Web server in the HTTP request. This is far more general, but of course uses a different mechanism from the GET method described earlier.

2.2. Persistent Connections

One of the aspects of HTTP that was noted after a few years of its use was that a lot of the time in sending and receiving a request was consumed in establishing the connection to the server. A Web browser might fetch a page containing references to k images (e.g. with the `<img src="name.jpg">`). When rendering this page, the client would have to fetch each of these images, and it would have to fetch the document and each of the k images in separate HTTP requests. The total download time for the "page", i.e. these $k + 1$ requests, was large because of the need to connect to the server $k+1$ times. Version 1.1 of the HTTP specification allows for persistent connections so that a client and server can maintain an open connection

without the need to re-establish it for each request. The use of persistent connections can yield dramatic speedups in the context of numerous short requests. Of course, there are a multitude of parameters controlling how and how long the connection remains open. The Web server needs to know when it can reasonably expect additional requests and communication and how long it should keep the channel open before "hanging up". Again, we want the option to control these settings, but not to require such information.

2.3. Secure Communication via HTTPS

We are familiar with submitting private information via HTML forms when, for example, buying airplane tickets or paying bills. And we also expect this information to be secure so that others cannot intercept it and commit identity theft. To this end, we use HTTP requests over SSL, a secure socket layer. SSL is a procedure developed by Netscape precisely for the purpose of submitting HTML forms securely, but that also applies far more generally to any communication using sockets. Essentially, HTTPS amounts to encrypted HTTP using asymmetric keys.

libcurl provides transparent access to secure HTTP using the https protocol qualifier in the target URI. User's don't have to specify any other details. However, they can specify a variety of different options controlling the use of SSL. For instance, they can specify the different version level of SSL support that is to be used in the different protocols. Similarly, there is support for specifying the location of the collection of trusted certificates to use when establishing a secure connection.

2.4. Authentication

HTTPS is concerned only with ensuring that others cannot see the content of the HTTP transmission. It is not concerned with verifying the identify of the client to the server. By sending a credit card number over HTTPs, I am not validating that I am the holder of the credit card, but merely ensuring that others cannot intercept and obtain the credit number via this communication. Authentication on the other hand involves the client and server verifying that the user is who she claims, or specifically that they have the right to use the resources based on how she identifies herself. This is typically done by sending an identifier and password of some form. There are several different mechanisms in use such as basic (plain text), digest, GSS-Negotiate (supporting Kerberos) and NTLM. Again, **libcurl**, and by transitivity **RCurl**, provides support these.

2.5. Cookies

Cookies are used in an HTTP conversation (or repeated requests to a server) by the client and server to provide state information across the requests. Cookies allow a server to send information in an HTTP response that the client will then send back in subsequent requests, thus preserving state. These are sometimes used to identify the client as being the same individual. A server might send a particular cookie when a user "logs in" and then use that cookie to lookup personalized information such preferences or billing information in subsequent requests. For security and privacy reasons, cookies are only sent by the client to the server that issued them. Thus, one has to manage the collection of all the cookies for different servers in the "cookie jar". Again, **libcurl** takes care of this for us but allows us to

control the details as we need.

Again, the details are not important for us in this paper. What is important is to understand that there are very many different options for HTTP both in sending a request and receiving a response. We need to have a way to control the entire communication, but also ignore the options that we don't need to control in any particular interaction. We need to be able to specify the header information for a request, include content within the body, and be able to process responses in flexible ways. And we want the underlying software to do as much of the detailed work as possible, such as escaping characters, dealing with chunked segments, handling details of passwords, cookies, etc. A list of features for which we need both transparent support and options to control the details include

- secure, encrypted connections,
- authentication and password files,
- management of cookies,
- proxy servers,
- customizable HTTP request headers,
- redirections for re-located or aliased URIs,
- different protocols such as FTP, LDAP,
- resuming downloads from a specified offset,
- creating connections with different timeouts, persistence, etc.

Few users will require more than a few of these features, but different users will require each of these as the range of applications increases.

3. Basic Functionality

The RCurl package provides an interface to the **libcurl** facilities. **libcurl** is, as the name suggests, a library programmed in C that provides portable tools for accessing URIs via HTTP requests and other protocols. It is feature rich, providing options for controlling almost all parts of the HTTP dialog. The RCurl package provides a high-level, more convenient access to the functionality in **libcurl** itself. In this section, we describe some of the RCurl functionality and the basic style of interaction. We will not cover in detail all the possible options that one can use. These are better left to the documentation pages of the R help files and **libcurl** itself. Users are encouraged to use the help facility in R and also to read the **libcurl** documentation available at http://curl.haxx.se/libcurl/c/

There are three high-level functions in **RCurl**: **getURL()**, **getForm()**, and **postForm()**. Each of these functions sends an HTTP request and expects a document in response. The functions differ in the type of document they retrieve and how, but each takes a URI and accepts a large and the same collection of options to customize the request and control the processing of the result.

3.1. Getting URIs

The most common use of HTTP is to download static or "fixed content" files. The function **getURL()** or **getURI()** provides a simple mechanism to do this. It takes the URL/URI in the usual form: **protocol://server/file/name**. The protocol will typically be http, but ftp - the File Transfer Protocol - is also supported by **libcurl** and hence **RCurl**. Similarly, HTTPS, FTPS, TFTP, GOPHER, TELNET, DICT, FILE, and LDAP protocols are supported (assuming the necessary supporting libraries are available). The server is the name of the Web server machine and can be given by name or IP address, e.g. 169.237.46.32. And, of course, we next identify the name of the file we want, giving the sequence of directory names separated by '/' symbols. If the Web server uses on a non-standard port (i.e. other than 80), one can specify the port after the ':' separating the host and file names, e.g. `http://www.omegahat.org:8080/index.html`.

We can fetch the main Web page for the RCurl package with the R command

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

The **getURL()** function uses the RCurl and **libcurl** facilities to send the HTTP request and receive the answer. It collects the body of the response into a single string and returns that. In our command, that string is now available in the variable `w`. It contains the source of the HTML page for the **index.html** file.

Note that the conversation between the client and Web server all takes place in exactly the way we described earlier. The client sends the GET request along with information in the HTTP header and then receives the response as a header and body. However, **getURL()** has hidden all of these details, and the default behavior does the correct thing in most cases.

In the simple call to **getURL()**, the return value is the content of the body of the HTTP reply, i.e. the contents of the requested document. This is is given as a single string, i.e. an R character vector of length 1. One can then process this value in whatever way is appropriate, e.g. read data using **read.table()** or **scan()**; parsing HTML or XML using **htmlTreeParse()** or **xmlTreeParse()** Temple Lang (2006d). **RCurl** has processed the response, identifying the necessary details from the response header and combining the body into a string whether it arrives in chunks or not. As we will see later, we can customize both the sending of the request and the processing of the response at various levels. However, we do not have to for basic requests.

3.2. Forms

There are two functions in **RCurl** that can be used to submit HTML forms via HTTP: **getForm()** and **postForm()**. These correspond to the two different ways of submitting a form: GET and POST. As we discussed in section 2, HTML forms allow the user to dynamically specify values for different input variables in the request via menus, checkboxes, etc. The client software encodes the variable name=value pairs in the HTTP request. For forms using the GET action, the name=value pairs are appended to the URI, separated by '&'. For forms using the the POST action, the name-value pairs are encoded in the body of the request. How this is done involves separating the sections of pairs of name and value lines by a boundary string which is identified in the request's header. This string must not occur in any of the names or values and so must be dynamically generated and validated. Fortunately,

these two functions hide all of the details of composing the query. For each, we just specify the URI of the form and then a collection of name-value pairs for the form variables given as a list or via the ... mechanism in R.

The Google search engine is perhaps the most widely known and used HTML form. We can programmatically send a query using the following R command[1]:

```
getForm("http://www.google.com/search",
         q ="RCurl",
         hl = "en", ie = "ISO-8859-1", oe = "ISO-8859-1",
         btnG = "Google+Search")
```

This submits the query with the single search word "RCurl". The other arguments provide context for the query to Googles search script giving the desired input and output encoding and language. The result is the HTML text that contains the search results in the usual format.

How do we find the names of the parameters and the appropriate values for the form submission? We can do this by examining the HTML source of the Google front page. Alternatively, we can take the information directly from the browser's URL bar after the query has completed. (We make use of the former approach in an other package **HTMLForms** to programmatically generate R functions that represent interfaces to Web forms and use **RCurl** to perform the queries.)

The **postForm()** has the same user-level interface and only differs in how the underlying HTTP request is formed.

3.3. Options controlling the request

At this point in our description, the functions **getURI()** and **getForm()** provide the same basic functionality that is already available in R, except that they add support for various different protocols such as HTTPS, FTPS and handle various aspects such as chunked responses. But the infrastructure provided by **RCurl** and **libcurl** allows us to customize each of these functions with numerous parameters that govern the HTTP request and response and how they are sent and received.

The RCurl package provides facilities at several different levels of complexity and control. The three functions in the simplest interface will suffice for most end-users. These allow the caller to download the contents of URI, and get or post a dynamic "form" request. For these, the only inputs that are needed are i) the URI of the document or script on the remote Web server, and, ii) for forms, the `name = value` pairs given as R arguments to the **getForm()** or **postForm()** functions. All the details of the HTTP dialog are handled by **RCurl** package and **libcurl**.

At the next level of complexity, the caller may want to specify additional information to govern the HTTP request. For example, some Web sites reorganize the files and directory structures and move files to different places on the Web server. This would causes problems for clients attempting to access the files that are no longer located in the original place. So the Web server can reply to the request that the page has moved and include the new location

[1]Note that one is not supposed to do this regularly using this approach, but rather use the Google programming interface.

as part of the header of the response. We could write R code to handle this response by examining the header and follow the "redirection". Fortunately, **libcurl** does this for us, and we can enable this facility using the *followlocation* option to its HTTP request mechanism. We can specify this option in our call to **getURL()** as a regular named argument (via the ... mechanism), or to **getForm()** and **postForm()** via the `.opts` argument. In the case of *followlocation*, we want to pass a value of TRUE or 1 (or some non-zero value) to enable that option. We can do this via either of the commands:

```
getURI( followlocation = TRUE)
getURI( .opts = list(followlocation = TRUE))
```

When submitting forms, one must use the `.opts` argument for specifying settings for the curl request. This is because the name-value pairs for the content of the form submission itself is specified via the ... argument and so we cannot mix the two sets of inputs via the variable arguments mechanism. So the `.opts` argument works as a way to specify options for the HTTP request for all the relevant functions in the **RCurl** package. For functions that accept curl options in both the ... and `.opts` forms, the two sets of options are merged into a single list of options. For an option that is present in both sets, the value in the ... will be used. This is convenient for interactive use when overriding values in a set of defaults options given as a single R object.

In addition to providing a single, consistent parameter across the different **RCurl** functions, using the `.opts` to specify a list of options is convenient when developing code that dynamically creates the list of options (rather than simply passing it from the caller to **RCurl** functions). Being able to pass the options as a single argument in the form of a list avoids programming gymnastics and indirect calls to put these back in the form of multiple, separate arguments to be handled via the ... mechanism. The `.opts` parameter is also useful when the function has a ... parameter of its own that is used for different purposes, e.g. specifying an arbitrary collection of URIs to download via the ..., with options controlling the centralized download in a different argument.

3.4. The Request Options

At the time of writing this article[2], there were 113 options that could be specified for a a curl request. These control a wide range of different aspects of how the request is submitted and how the response is processed as it is received. These correspond to the C-level options described in the **libcurl** documentation pages Stenberg and the cURL development team (2006). The names in R are direct mappings from the C-level names using the following rule. In R, the `CURLOPT_` prefix is removed, the word is converted to lower case, and underscores (_) are replaced by periods (.). For example, `CURLOPT_NETRC` in C becomes netrc in R, and `CURLOPT_NETRC_FILE` in C becomes netrc.file.

The names of all the options are dynamically available via the R function **listCurlOptions()**. The options fall into several categories (taken from the **libcurl** documentation):

[2]Version 7.15.3 of **libcurl**.

Behavior	controls diagnostic output, whether process-level signals are handled by **libcurl**;
Network	target URI, proxy information, network interface and IP resolution parameters;
Connections	controls for the socket connectivity between the client and the server;
Authentication	passwords
HTTP & FTP	parameters controlling the behavior of sessions using these protocols;
SSL	facilities for controlling the use of SSL and digital certificates;
Callbacks	event handler functions for dynamically interacting with **libcurl** (see next section).

We will not describe all the options here in this paper as they will become outdated and will merely be a duplicate of the documentation accompanying **libcurl** and on its Web site.

Generally, there is a one-to-one mapping between the options available in **libcurl** and those specifiable in **RCurl**. There are certain ones which don't make as much sense for the R interface such as specifying the error buffer which **libcurl** will use for making human-readable error messages available in the event of an error. This is a C-level data structure and while we can provide an interface to it from R, the **RCurl** package automatically handles reporting errors.

While there are many options, they are easily grouped into different categories such as HTTP, FTP, network, authentication and this is done in the **libcurl** documentation. Perhaps the group that needs most explanation in the context of R is the collection of callbacks.

3.5. Callback Options

The value of so-called "callback" options are typically specified in R as an R function. These functions, if specified, are then invoked by **libcurl** when certain events occur that correspond to that option. In essence, we are passing a function to **libcurl** which "calls back" to R by passing control to that particular function, providing it some inputs which give context to the particular event.

There are several callback options in **RCurl** corresponding to different events in **libcurl**. These are

1. *writefunction*,
2. *headerfunction*,
3. *debugfunction*,
4. *progressfunction*,

We will give a brief description of how to use each of these.

writefunction When **libcurl** processes a response from a request, it reads the header and then makes the body of that response available to the caller by invoking a C routine or function specified as the value for the *writefunction*. It is called *writefunction* as, to **libcurl**, it needs to write the blocks of text in the body to a particular place. For each block of text **libcurl** receives, it calls the function and provides the text of this part of the response as a string. In **RCurl**, we specify an R function for this option. Each time **libcurl** has data from the response, it calls this function. It is called with a single value which is the string giving the current segment of the response. The function can have additional arguments, but these must have default values.

The R function given as the callback for the *writefunction* option is expected to return either a logical value indicating whether it was successful (`TRUE`) or not (`FALSE`). Alternatively, it can return the number of characters that were processed. If this is not the same as the number passed to it, **libcurl** will raise an error and terminate the request. Alternatively, if the content is not meaningful to the R function or causes problems, the function can raise an error itself via the R function **stop()**. Thus there are two different ways to stop the processing of a request: an error in the R callback function, or simply returning `FALSE`. The former does not return control back to the **libcurl** routine that invoked the R callback and so **RCurl** does not have an explicit opportunity to cleanup the connection to the Web server, etc. at that point. Returning `FALSE` does provide **libcurl** with an opportunity to exit more gracefully. Both approaches are appropriate in different circumstances.

The function **basicTextGatherer()** in the **RCurl** package provides a generic and simple implementation for collecting the text of the body of a response. When one calls **basicTextGatherer()**, it returns a list containing three functions. The *update* element can be supplied as the value for the *writefunction* callback option. When the processing of the response has been completed, a call to the function in the *value* element of the list will return the full text of the response. This is illustrated with following simple sequence of commands:

```
g =  basicTextGatherer()
getURI("http://www.omegahat.org", writefunction = g)
g$value()
```

If we want to accumulate responses from a series of calls, we can use the same instance of this list of functions returned from a single call to **basicTextGatherer()**. Also, we can use the same function objects but clear existing text by calling the *reset* element of the list to reinitialize the text to the empty string and discard previously collected text. And of course, we can create separate instances of these functions with separate calls to **basicTextGatherer()** and use these for processing different responses independently of each other.

Note that these functions will not be able to handle binary data. We use C routines for that currently which can be specified from R via the *writefunction* in a similar manner (see below).

The **basicTextGatherer()** is, as the name suggests, quite basic. It merely combines the different blocks as they are passed to the function from **libcurl**. It is entirely possible to do more interesting processing of the content at this point. For example, we could extract data values from the text and add them directly to a vector or data frame as we receive them. This avoids accumulating the text and then at the end of the request converting that into the values in one large operation. This allows us to avoid the overhead of having two copies of the data in memory when we convert it, i.e. the text and the extracted values.

We can also specify a function for the *writefunction* option that filters content as it is encountered. For example, if the request consists of lines of a Web log file, we might only keep those records that satisfy a certain criterion, e.g. requests for a specific page,

or in a particular period of the day. This form of asynchronous callbacks is thus similar to streaming data.

headerfunction The *writefunction* is used by **libcurl** to pass the content of the body of the response back to the client. By default, this content does not include the header information of the response. We can arrange for **libcurl** to combine the two using a non-zero value for the *header* option. However, then we end up with two separate pieces of text that we have to manually separate. Instead, we can use the *headerfunction* in much the same way as the *writefunction* to provide an R function that will be invoked with the contents of the response header only. This allows us to gather the header and body separately.

Since the header is typically quite short, it is common that the callback function will only be invoked once with the entire text of the header. However, this is not guaranteed. One can make use of the **basicTextGatherer()** function again to collect the text across the calls to the callback function. The function **parseHTTPHeader()** is available to then convert the entire text of the header into the name-value pairs and also process the first line of an HTTP request and return the action, status code, etc. The function **basicHeaderGatherer()** is a marginally more convenient variant of these two steps. The *value* function it returns will automatically call **parseHTTPHeader()** and return the value. This can be used as

```
d = basicHeaderGatherer()
body = getURI( "http://www.omegahat.org/RCurl", headerfunction = d)
d$value()
```

Being able to process the header dynamically before the body of the response is retrieved allows us to collect information from the header settings that will allow us to prepare for processing the body via the *writefunction*. For example, we can determine the type of content - simple text, HTML, XML or binary data - and communicate this to the *writefunction* callback function. Or we might be able to determine the size or type of the data structure for the body and preallocate it in order to save memory when processing the body

debugfunction Occassionally, requests will fail or return data that seems wrong or incomplete. One can, and should, enable **libcurl**'s diagnostic mode via the *verbose* option which causes it to output information to the console about what it is doing, showing the request and the response headers, and more. To get even more information, we can specify a callback function for the *debugfunction* option which **libcurl** will then use when it has information to share. It calls this with three arguments: the message text; the "topic" of the notification which is one of text, headerIn, headerOut, dataIn and dataOut; and a reference to the curl object performing the request. The function **debugGatherer()** is a simple version that collects the information from the different types in their respective collections. This removes the order, but makes it easy to see what happened in the different streams. R users can specify other functions to provide different information, compare messages with expected values in the inputs to the request, and so on.

progressfunction Some requests may take a considerable length of time to complete. It can be helpful to be notified about the progress of the request. We can then provide

feedback to the user and tell them how long they might have to wait. We can display a message on the console or update a graphical user interface, for example, although this requires interaction with an event loop to update the GUI display.

This callback option allows us to specify an R function that will be called at regular intervals while **libcurl** is processing the request. It is called with two arguments providing information about the number of bytes for both upload and download. Each argument is an integer vector with two elements: the total for the request and the number of bytes up to this point in the processing of the request. When specifying a callback for this option, one must also set *noprogress* to `FALSE`.

C routines and R functions are quite similar in concept. Typically, R users will provide an R function as the value of a callback option in **RCurl**. These are relatively easy to write and modify, and using closures allow one to manage mutable state across successive calls. However, for efficiency and flexibility reasons, it is also useful to be able to specify the address of a C routine and have that be used directly. This avoids converting data from C to R for the function call, and sometimes this is important, e.g. when dealing with binary content. Additionally, it allows for the reuse of existing code in C libraries.

While most uses of callback options will provide an R function as the action, we can also specify compiled routines which will then be invoked by **libcurl** in the same manner as the R functions. In R, we can explicitly reference C routines in DLLs that have been loaded during the session. The function **getNativeSymbolInfo()** takes the name of the routine and the name of the DLL (dynamically loadable library) in which it is located and returns information about that routine. This information includes its address in memory. This object can be passed to the curl handle as the value of the particular option. The **RCurl** compiled code is set up to differentiate between a function and a native routine and handle them appropriately. Let's suppose we had written a routine named `readBinaryData` which reads a particular form of binary data, e.g. a particular type of micro-array gene expression data, and that we have dynamically loaded the compiled code into R from the DLL named `myDLL`. To establish this as the routine that **libcurl** will call as it receives the response from the request, we can use the command

```
getURI("http://....",
       writefunction = getNativeSymbolInfo("readBinaryData", "myDLL")$address)
```

This facility is especially appropriate for the *ssl.ctx.function* and the *readfunction* options as the callback functions may have to work with low-level C data structures.

Precisely how we retrieve the result after **libcurl** has completed the request is up to the author of the native routine. One might use a global variable in the native code and have a routine that could be called from R to access the data after the request finishes. The **libcurl** programming interface also allows one to set the callback routine along with a corresponding DATA option, e.g. WRITEDATA. The value provided in the DATA option is passed to the callback routine when **libcurl** invokes it. It therefore acts as a way to parametrize a single routine for use in multiple contexts. We use this option internally in **RCurl** to associate the R function with the internal callback routine. These options are not for use in the **RCurl** package when specifying an R function as a callback. Note however that no attempt is made to prevent one setting these options. They are useful when specifying C routines as callbacks

and associating a particular instance of a data structure that is specific to this request. This allows us to avoid the use of global variables.

There are several other callback options, specifically *readfunction*, *ioctlfunction*, *ssl.ctx.function* and three conversion functions. Currently, the **RCurl** package does not allow one to specify an R function as the value for these. For each of these, one can specify a C routine to be called and use the corresponding DATA option to add a value that will be passed to the routine. These options are only amenable to C routines at this point as they deal with low-level native data structures in C which currently do not have R interfaces. This might change with the ability to programmatically generate the bindings to the relevant libraries. Fortunately, we typically do not need to supply callbacks for these options as the computations can be preprocessed in R. For example, the *readfunction* is used when **libcurl** requires input that is to be sent to the server as part of the request. This might occur, for example, when uploading a file. In this case, the caller can read the contents of the file directly into R and pass it as a string via the *postfields* option.

3.6. The Computational Model in RCurl

Aside from the three high-level functions – **getURI()**, **getForm()** and **postForm()**, the interface presented in **RCurl** mirrors that provided by **libcurl** with some facilities added to make the programming easier for the R programmer than those using the C interface. **libcurl** provides an opaque data for performing HTTP requests. In **RCurl**, we refer to these as "curl handles". Each curl handle has a collection of options or settings that it uses to perform an HTTP query. Each of these settings persists within the handle until it is updated with a new value or, of course, the handle is discarded and garbage collected by R. These settings allow us to control very many aspects of how the handle process the HTTP request and response. The low-level model is that we set the the options of interest such as the target URI, new fields to be added to the header, user name and password, location of SSL certificate files, callback functions to read and write content involved in the query, etc. Most queries will involve setting only a few of the 113 supported options. Once the options are set, one instructs the handle to issue the query and get the response. In the functions **getURI()**, **getForm()** and **postForm()** as with several other functions in the package, a curl handle is created implicitly within the function call (as a default value for the `curl` argument) and persists only for the duration of that function call.

We can create a new, default curl handle using the function **getCurlHandle()**. This generates a new **libcurl** data structure, sets the options to their default values and returns an opaque reference to the internal handle as an R object. Unlike the **libcurl** API, we can also set options in the new handle in the same call that creates it, e.g.

```
h = getCurlHandle()
h = getCurlHandle(followlocation = TRUE)
```

We can also create a new curl handle by duplicating an existing one using the function **dupCurlHandle()**. This is useful if we have set options in an existing handle and want to continue to use it but also want to make requests with a slightly different set. Rather than creating a new handle and re-setting the options we previously had to specify, we can inherit them from our original handle. And we can also override settings in the new handle within the call to **dupCurlHandle()** via the ...mechanism and `.opts` parameter.

The options remain in effect for a given curl handle across requests and until that handle is no longer in use. So we can, for example, merely set the target URI to a new value via the *url* option and perform a different request with all the same settings as before. For example, the following code downloads 2 documents in succession using the same handle:

```
h = getCurlHandle()
getURI("http://www.omegahat.org/RCurl/index.html", curl = h)
getURI("http://www.omegahat.org/RCurl/Changes.html", curl = h)
```

This saves us (implicitly) creating a new handle for each request and setting all the shared options repeatedly. Additionally, if the curl handle is set to use persistent connections and (a subset of) the series of requests are to the same server, there can be significant performance gains by reusing the same handle and avoiding the overhead of reconnecting to the server. Using the same handle across requests is not necessary in usual operations since **getURI()** is vectorized and will work on multiple documents, but the basic point of creating a handle and using it in multiple operations is generally important for advanced use.

The **RCurl** package simplifies the sequence of steps inherent in the **libcurl** API, primarily because of the default argument mechanism in R. Rather than creating the handle, setting the options, and performing the request as three separate actions, one can combine them into a single operation. The functions **getURI()**, **getForm()** and **postForm()** allow us to specify the options and the curl handle as inputs, with suitable defaults for those that are not specified, and these functions perform the HTTP request and processing of the response. These three functions are based on the slightly lower-level function **curlPerform()** which, as the name suggests, actually performs the request. It is the bare interface that takes a handle and any curl options and performs the query. This is the most flexible interface to the basic **libcurl** facilities and allows us to reuse existing handles or implicitly create a new handle for the particular request. The example of downloading the two URIs can be done with **curlPerform()** as

```
h = getCurlHandle()
curlPerform(uri = "http://www.omegahat.org/RCurl/index.html", curl = h)
curlPerform(uri = "http://www.omegahat.org/RCurl/Changes.html", curl = h)
```

This does not appear any simpler, but when we have requests that are not simple GET and POST operations, **curlPerform()** is needed to create and send the request.

curlPerform() marshals any options to the handle using the other **RCurl** function **setCurlOpt()**. This function allows R users to explicitly update options settings in an existing curl handle without actually performing a request. These new settings will be used in subsequent requests that use this handle.

In some more rare situations, it is useful to be able to specify a set of options to be used by a handle, but without actually setting them in the handle. Rather, we want an R representation of the options. **curlOptions()** is a function that creates such a list of R-based curl options and validates them by matching the names again those understood by **libcurl**. These can be saved between R sessions and used in a call to **curlSetOpt()** to "restore" an existing handle. Unfortunately, this is the only way to do this as when a curl handle is serialized in R, the reference to the opaque **libcurl** data structure cannot be de-serialized or restored in a new R session. And also, the **libcurl** software does not provide a way to query any or all of the

current values of an curl handle's options. As a result, we cannot retrieve them and store them in R to be reused later.

When a request has been processed, **libcurl** stores information about the processing of that request in curl handle. This includes information such as the effective URI (after redirects, etc.), the status of the request, the total time and the times for the different sub-tasks, the content type and size, etc. Given the curl handle used to make the request, this information can be retrieved by calling **getCurlInfo()**. For example,

```
h = getCurlHandle()
getURI("http://www.omegahat.org/RCurl/index.html", curl = h)
names(getCurlInfo(h))
```

```
 [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"
```

This is useful in several ways. For one, it allows us to change subsequent repeat queries to the actual URI if it is different from the nominal URI. Also, it allows us to measure the different characteristics of our communication and potentially dynamically understand the operating characteristics of our software that uses the HTTP requests and optimize it, if this is relevant. Of course, we have to create the curl handle separately rather than rely on a default handle being created so that we can pass that same handle to **getCurlInfo()**.

3.7. Multiple Requests and Handles

As we have described, we can create numerous separate curl handles to process different requests and reuse a handle for a succession of multiple requests. This gives us a lot of flexibility in programming network requests in R. However, when we perform a query in R either directly using **curlPerform()** or a higher-level function, control is passed to **libcurl** and R must wait until that request has completed. If we have multiple documents to retrieve, we must do them sequentially if using the curl handles as described earlier. However, it is easy to see that if a Web server is responding slowly, or if resolving the Internet Protocol (IP) address of a server from the given name takes a long time because of a slow domain name server (DNS), then R and **libcurl** can potentially spend a lot of time idle. If **libcurl** could be given several requests at once, then it could send them concurrently and then check back on each of them to see how they were progressing. A slow Web server would not inhibit the speed at which the other requests were being processed. This is a form of multi-tasking with which we humans are familiar. It does not necessarily involve a multi-threaded programming

model (although **libcurl** supports that), but rather an ability to manage multiple requests at the same time and be able to interleave the processing of these requests. **RCurl** provides an interface to **libcurl**'s "multi" interface for processing multiple requests concurrently.

This multi-request interface uses much of what we have discussed so far. Essentially, we create each of the different concurrent requests in much the same way as before by creating a regular curl handle via **getCurlHandle()** and setting the options in that call or in subsequent calls to **curlSetOpt()**. At any point in time, each of these curl handles will process a single request. (We can reuse a handle when its request is complete.)

Next, we create a `"MultiCURLHandle"` object that can manage these different requests using the function **getMultiCurlHandle()**. And we add each of the simple curl handles that we want processed to this multi-handle. We add these using the generic function **push()**. When we have added the initial set of request handles to the multi-handle manager, we can call **curlMultiPerform()**. This then hands control to **libcurl** which processes the requests. When any one of these requests is complete, it returns control back to the R caller, telling how many requests still remain to be completed. This gives us an opportunity to do something in response to the completion of the request. Alternatively, if we just want all the requests to be completed, we can call **curlMultiPerform()** with the *`multiple`* argument given as `TRUE`, or more conveniently simply call the generic **complete()** function. This will then only return when all the requests have been completed.

Just as we can add regular curl handles to the multi-handler manager, we can also remove them at any time. We use **pop()** for this. Both **pop()** and **push()** return an updated R object containing the managed curl handles. So it is important to reassign that value to an R variable, typically the one containing the original value of the curl multi-handle.

Also, when a request associated with a particular handle is complete, we can reuse the handle to issue another request. It is simplest to first **pop()** that handle and then set the options for the new request and **push()** it onto the multi-handler again.

This multiple request interface is very powerful. We will see how to use it in more detail in the case studies in section 5.

3.8. Additional RCurl Functions

It is often useful to know which version of curl, etc. we are using. The function **curlVersion()** provides information about the version of **libcurl** installed on the particular machine. This reports on the version of the **libcurl** library, the type of host, the features supported in this particular installation of **libcurl** such as IPV6, SSL, support for largefiles, etc. It also lists the supported protocols and whether asynchronous host name resolution is used. An example of the output is

```
curlVersion()
$age
[1] 2

$version
[1] "7.15.3"

$vesion_num
```

```
[1] 462595

$host
[1] "powerpc-apple-darwin8.6.0"

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

$ssl_version
[1] " OpenSSL/0.9.7i"

$ssl_version_num
[1] 0

$libz_version
[1] "1.2.3"

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

$ares
[1] ""

$ares_num
[1] 0

$libidn
[1] ""
```

The result is a list with 12 named elements detailing the different information. This is a regular R object. This information can be used by high-level functions to determine how to perform a query. For example, if secure HTTP (HTTPS) is not supported, an alternative approach may be used.

The package also provides some miscellaneous functions that are useful for dealing with HTTP requests. As we mentioned when describing HTTP, certain characters in a URI need to be transformed into a different representation. Non-letter characters and so-called "foreign" characters cannot be uniformly expressed in a string on all different computer systems as simple characters. Before being sent in a request, the function **curlEscape()** can be used to transform a string to its "escaped" form with the non-character elements translated to their hexadecimal representation. And **curlUnescape()** can be used to map such an "escaped" string back to a regular character string.

4. Examples

In this section we take the opportunity to illustrate some of the options that **libcurl** supports and how to use them via **RCurl**. It is in no way meant to be a comprehensive illustration of all the options. Again we refer the reader to the help pages for the package and the documentation for **libcurl**for more details.

4.1. SOAP

The Simple Object Access Protocol (SOAP) Snell *et al.* (2002) is a mechanism by which function calls are sent over HTTP to remote servers. This is the primary mechanism underlying Web services. It allows us to use functionality in remote servers from within different applications. It is similar to RPC (Remote Procedure Call), Microsoft's DCOM (the Distributed Component Object Model), CORBA (the Common Object Request Broker Architecture) and Java's RMI (Remote Method Invocation). What differentiates SOAP from these other approaches is that it uses HTTP as a simple way to deliver the request and response, even through firewalls. Additionally, it uses XML to encode the call and the data. To allow R to be a client of a SOAP server and utilize remote SOAP servers to retrieve data or perform computations, we need the ability to send HTTP requests in R. And this was one of the primary motivations for developing the **RCurl** package.

The details of a SOAP call become lengthy, tedious and complex. These are hidden from R users via functions in the **SSOAP** package Temple Lang (2006c). However, it may help to illustrate the basic mechanism that uses **RCurl**. We will do so with a very simple test example as we do not want to focus on the details of SOAP and XML, but rather how to use **RCurl** to make the request.

Our example queries the xmethods web service to get the currency exchange rate between the U.S. dollar and the Irish punt. The service is located at `http://services.xmethods.net/soap` and we are calling the `getRate` method. We will pass the names of two countries - Ireland and USA - as the two arguments.

SOAP makes requests by using the PUT operation in HTTP, similar to the way POST forms are sent. The actual request is sent as XML text as the body of the request. In spite of the similarity to posting forms, we cannot use the **postForm()** function as that expects `name = value` arguments giving the inputs to the form. Instead, we use **curlPerform()** directly and specify the different elements of the request as different options for the curl handle. We thus have to provide both the body of the request and additional fields for the HTTP request header.

The body of the HTTP request for this method call should consist of the following XML text:

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
 <SOAP-ENV:Body>
   <namesp1:getRate xmlns:namesp1="urn:xmethods-CurrencyExchange">
      <country1 xsi:type="xsd:string">Ireland</country1>
```

```
      <country2 xsi:type="xsd:string">USA</country2>
   </namesp1:getRate>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The **SSOAP** package would create this text for us, so we need not concentrate on the content and can assume that it is available in the R variable `body`, say, as a single string.

The body of the POST operation is given as the value for the *postfields* option in curl. This is all that is needed in the body, but to make the request work however, we need additional fields in the header which are mandated by SOAP.

The outgoing header should end up looking something like

```
POST /soap HTTP/1.1
Host: services.xmethods.net
Accept: text/xml
Accept: multipart/*
Content-Type: text/xml; charset=utf-8
SOAPAction: urn:xmethods-CurrencyExchange#getRate#getRate
Content-Length: 590
```

We specify the server URI as `http://services.xmethods.net/soap` and **RCurl** will generate the first two lines of the header from this by splitting the host name and document path. We must also specify the two Accept entries, the Content-Type and SOAPAction fields. These are specific to our request and not something RCurl can determine. The Content-Length field refers to how many bytes there are in the body of the POST request and, of course, **RCurl** can determine this from the body before sending the request. The Content-Type field indicates to the Web server how the body is formatted. The Accept fields tell the Web server what formats are acceptable for its response. The SOAPAction field is used to direct the request through the Web server and onto the SOAP server.

We specify the Accept, Content-Type and SOAPAction header fields via the *httpheader* argument for **curlPerform()**. The value should be a named character vector with each name element identifying the header entry, and the value of the element giving the value for the header field.

At this point, we can put the R command together as

```
headerFields =
   c(Accept = "text/xml",
     Accept = "multipart/*",
     'Content-Type' = "text/xml; charset=utf-8",
     SOAPAction="urn:xmethods-CurrencyExchange#getRate#getRate")

curlPerform(url = "http://services.xmethods.net/soap",
            httpheader = headerFields,
            postfields = body
           )
```

curlPerform() merely returns the status code of the operation indicating whether it was successful (0) or not. Instead of this, we want the actual body of the response from the web

server. To get this, we need to provide an R function to collect that text. We do this with the *writefunction* option for the handle. We can use the **basicTextGatherer()** in the **RCurl** package for this. This function returns a list of functions that share mutable state. The *update* element of the list it returns is the function we pass to **curlPerform()**. When we want to retrieve the updated state, we invoke the *value* element of the list which is a function that simply returns the current contents of the shared string used to gather the text collected by the update function. So our command is

```
reader = basicTextGatherer()
curlPerform(url = "http://services.xmethods.net/soap",
            httpheader = headerFields,
            postfields = body,
            writefunction = reader$update
           )
reader$value()
```

The string returned from the **value()** function contains the result, but encoded in XML and SOAP. So further processing is necessary.

curlPerform() will claim success if it did its task correctly, but that does not guarantee that the HTTP request was successful. To determine this, we must look at the header of the response. To get access to this, we can use either of two approaches. Firstly, we can set the *header* in the curl options to `TRUE` and then the header will be relayed to the write-function as the first part of the body. A better approach is to collect the text from both the body and header separately. We supply an R function for the *headerfunction* argument of **curlPerform()**. Again, we can use **basicTextGatherer()**.

```
reader = basicTextGatherer()
header = basicTextGatherer()
curlPerform(url = "http://services.xmethods.net/soap",
            httpheader = headerFields,
            postfields = body,
            writefunction = reader$update,
            headerfunction = header$update
           )
```

We can then look at the text of the response header and determine how to process the body. The function **parseHTTPHeader()** breaks the header content into a named list giving the values of the fields. It also processes the first "status" line and includes the return status as an integer in the list of header fields. This allows us to tell whether the call succeeded or not, i.e. had a return status between 200 and 300.

```
h = parseHTTPHeader( header$value() )
h$status >= 200 && h$status < 300
```

If we are simply interested in the return status of the HTTP request and not the general contents of the response header, we can use the **getCurlInfo()** function to query that. We call this with the curl handle used in the processing of the request. Therefore we have to create the handle and pass it explicitly to **curlPerform()** so that we can also pass it to **getCurlInfo()** after the request has been completed. We would do this as

```
reader = basicTextGatherer()
handle = getCurlHandle()
curlPerform(url = "http://services.xmethods.net/soap",
            httpheader = headerFields,
            postfields = body,
            writefunction = reader$update,
            curl =handle
           )
status = getCurlInfo( handle )$response.code
```

We emphasize that there is a much higher-level interface to SOAP client services within R via the **SSOAP** package. It hides all the details of the actual call and can also automatically generate R functions that provide an interface to a server's methods and data structures by using reflectance. The underlying code uses **RCurl**, but users do not need to know how. The purpose of our example is to illustrate the power and flexibility of the **RCurl** interface for non-trivial requests.

4.2. Authentication and Passwords

Some Web sites require a user name and password to access certain files. We can specify this in a call via the *userpwd* option for curl. This is a simple string that gives the user name and password in the form `user:password`. The Web site for the **RCurl** provides a test for this. The user name is 'bob' and the password is 'welcome'. So we can request the page with the R command

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

Some people like to keep passwords in a 'netrc' file, often in their home directory in a file named **.netrc**. **libcurl** can read such files and use the passwords from there. We have to tell it to use such a file via the *netrc* option. We pass a value of 1 which indicates that the file is optional - if it is present, **libcurl** will read it to find the password, however we can also specify the password via the *userpwd* option to override values in that file. Assuming we have the file **.netrc** in our home directory and it contains the line

```
machine www.omegahat.org  login bob  password welcome
```

we can use the R command

```
getURL("http://www.omegahat.org/RCurl/testPassword/index.html",
       netrc = "optional")
```

to access the restricted page. Note that we used the symbolic value "optional" rather than the C-level value 1 for the option. We can use either and the **RCurl** package coerces the value to the associated enumerated value.

If the password file is not in a standard location, we can use the *netrc.file* option to specify its full path:

```
getURL("http://www.omegahat.org/RCurl/testPassword/index.html",
       netrc = 1,
       netrc.file = "/Users/duncan/.netrc")
```

libcurl attempts to determine what the authentication mechanism is used by the server. There are several (Basic, Digest, GSS, NTLM) and we can specify the one to use via the *httpauth* option. The best option is ANY or ANYSAFE.

4.3. SSL

We can check whether the particular installation of **libcurl** and the **RCurl** package supports https requests with the command

```
"https" %in% curlVersion()$protocols
```

If this returns `TRUE`, then it does have support for this facility. And then, typically, one need only use the 'https' qualifier for a URI to use this encrypted connection.

For some Web servers, the SSL layer will fail to connect to the Web server because it cannot authenticate that server. SSL uses digitally signed certificates to verify the identity of the server so as to avoid sending sensitive data to bogus servers. **libcurl** provides options such as *sslcert* to control where the SSL library looks for local certificates that validate a server. In some cases, we may want to trust a server is authentic even though we do not have a certificate. For example, to access the Subversion repository for R, we need to avoid validating the host. We use the two curl options *ssl.verifyhost* and *ssl.verifypeer* and set them both to `FALSE` to turn off the verification. We also specify the *followlocation* option to allow **libcurl** to follow any redirections to other URIs by the remote server. So the command to access this page is then

```
getURI("https://svn.r-project.org/R/trunk",
       ssl.verifyhost = FALSE,
       ssl.verifypeer = FALSE,
       followlocation = TRUE)
```

There are numerous options that control how the SSL layer connects to the server. Some of these are technical and require a good understanding of SSL and so are outside the scope of this paper. Readers are referred to both the **libcurl** and openssl (www.openssl.org) documentation for more information.

4.4. Cookies

As we mentioned in section 2, cookies are used by the server to maintain state across separate transactions with a client. These allow the server to "remember" you in subsequent interactions. The server sends cookies as *name=value* pairs in the header of the response (as Set-Cookie instructions) and the client is expected to manage these and include the cookies from the target server in subsequent requests. Cookies have an expiration date and also a path and domain (or host name). A client is supposed to discard a cookie after the expiration date and replace it with a new instance if the server sends it one. Also, the server can specify the domain and path for a cookie which associates the cookie with a particular document

or part of the Web site. The client is supposed to send the cookie with any request for a document within the domain and path. As with many aspects of HTTP, the details of a simple idea can become complex to manage and so we leave it to **libcurl**.

By default, **libcurl** ignores cookies, but we can activate the management and transmission of cookies by setting one of several options in a curl handle, specifically setting either of the *cookiefile* or *cookiejar* options. These are quite different, but both cause the curl handle to recognize and manage cookies.

The *cookiefile* option tells the curl handle to read a collection of cookies from the specified file. If the file does not exist or is empty, this only has the side effect of making the handle process cookies. If the file does contain a collection of cookies, these are read and used in subsequent requests. This allows us to store cookies in a file and use them in a different R session. The format of the file is described in the document http://www.netscape.com/newsref/std/cookie_spec.html and looks something like

```
# Netscape HTTP Cookie File
# http://www.netscape.com/newsref/std/cookie_spec.html
# This file was generated by libcurl! Edit at your own risk.

.nytimes.com TRUE / FALSE 1183206005 RMID 8362050d3d9744a516f5a350
.google.com  TRUE / FALSE 2147368447 PREF ID=fd3870ecb4a92b03:TM=11516700
```

Some applications store cookies in an XML format.

The *cookiejar* option gives the name of a file into which the handle will write its collection of cookies, but only when it is being discarded and is no longer in use. The idea is that **libcurl** will manage the cookies for the lifetime of that handle without our intervention, accepting new cookies in responses and sending them in subsequent requests. When the handle is no longer in use, it serializes the known cookies to a file so that they can be read in a future session or by a new handle. Unfortunately, we cannot query the known cookies in a handle while it is still in use. To access these, we must discard the handle and cause it to be garbage collected by R by removing any reference to it (and explicitly calling the garbage collector via **gc()**). When the handle is reclaimed, **libcurl** arranges to write the cookies to the specified file.

Regardless of whether **libcurl** is managing cookies or not, one can always manually add one or more cookie name-value pairs to a request using the *cookie* option. The cookie value is given simply as a string of the form `name=value`. And we can specify multiple cookie values in the string by separating them with the ';' character. For example, let's suppose that you have registered with the NCBI PubMed site (http://www.ncbi.nlm.nih.gov/entrez) and received an account. At the end of the registration process, the site will typically have sent a cookie in its response that uniquely identifies you (at least on that machine). Your browser will be using that in each transaction it has with that site and that is how the Web site personalizes its pages for you. Using your browser, you can examine the collection of cookies and hopefully identify the pertinent one. In the case of PubMed, we are looking for the cookie named `WebCubbyUser` within the nih.gov domain. We can then pass its value, say ABC1234, in our R-based request with the command

```
getURI("http://www.ncbi.nlm.nih.gov/entrez",
```

```
cookie = "WebCubbyUser=ABC1234",
followlocation = TRUE)
```

The resulting document will contain your personal login for that site, illustrating that the cookie was communicated correctly.

5. Advanced Usage

In this section, we will present some more detailed and sophisticated case studies in using **RCurl**. These mainly focus on the use of the asynchronous "multi"-request interface. In the first example, we explore how we can process the response of a request "on-the-fly" as it is being received in chunks in order to improve the processing efficiency. Next, we look at how we can connect the output from a **RCurl** request as input to an XML parser so that the parser queries more input from the HTTP request as it is needed. We then focus on making numerous simultaneous requests that are processed concurrently. And finally, we illustrate how we can recursively download all linked documents from within a given document in an efficient manner.

5.1. In-line processing

The basic use of **getURL()**, **getForm()** and **postForm()** returns the contents of the requested document as a single block of text. It is accumulated by the **libcurl** facilities and combined into a single string. We then typically traverse the contents of the document to extract the information into regular data, e.g. vectors and data frames. For example, suppose the document we requested is a simple stream of numbers such as prices of a particular stock at different time points. We would download the contents of the file, and then read it into a vector in R so that we could analyze the values. Unfortunately, this results in essentially two copies of the data residing in memory simultaneously. This can be prohibitive or at least undesirable for large datasets.

An alternative approach is to process the data in chunks as it is received by **libcurl**. If we can be notified each time **libcurl** receives data from the reply and do something meaningful with the data, then we need not accumulate the chunks. The largest extra piece of information we will need to have is the largest chunk. In our example, we could take each chunk and pass it to the **scan()** function to turn the values into a vector. Then we can concatenate this with the vector from the previously processed chunks.

The `write` parameter of the **getURI()** and other **RCurl** functions allows us to do just this. We can give it an R function that is called each time **libcurl** receives data from the HTTP response from the server. The function is called with a string as the only argument. The string is the current chunk of data received by **libcurl**. The function can do whatever it wants to the content. The function can return a logical value indicating whether it was successful (`TRUE`) or not (`FALSE`). Any other type of value returned is ignored. This allows the function to terminate the processing gracefully.

The parameter is called `write` corresponding to the **libcurl** option `writefunction`. This might seem confusing as essentially we are reading data. However, from **libcurl**'s perspective, it is writing it somewhere.

In our example, we might write the chunk processor function as something like the following:

```
function(chunk)
{
  con = textConnection(chunk)
  on.exit(close(con))
  tmp = scan(con)
}
```

This arranges to call **scan()** on the text given by `chunk`. It is unfortunately slightly more complicated as we have to tell **scan()** that the string is not the name of a file, but the text containing the values. We use a `"textConnection"` object for this purpose and must arrange to close that connection when we have finished with it.

The only difficulty with the function we have written is that we have not added these values to those of the previously processed chunks. We haven't defined a variable that stores these earlier values. Unfortunately, we cannot return these new values and expect **libcurl** to know what to do with them. Instead, we could use a global variable, but this is rarely desirable. For example, if we want to read data in this manner recursively, a global variable would be overwritten for each request. And if there is an error in processing one of the chunks, the global variable must be cleared or remain in a confused and confusing state.

In R, we can use closures, also known as lexical scoping, to keep the vector from the earlier chunks in a local variable that persists across the (asynchronous) calls. This is described in Gentleman and Ihaka (200) and R Development Core Team (2005). The basic mechanism is to define functions within a function. The interior functions share state and variables defined in their common environment. So we can define one function that processes the chunks and stores the values and another that provides access to the results at the end.

```
stockReader =
function()
{
  values <- numeric() # to which the data is appended when received

   # Function that appends the values to the centrally stored vector
  read = function(chunk) {
            con = textConnection(chunk)
            on.exit(close(con))
            tmp = scan(con)
            values <<- c(values, tmp)
          }


  list(read = read,
       values = function() values  # accessor to get result on completion
      )
}
```

The outer function - **stockReader()** - is called each time we want to create a new reader function with its own, separate `values` vector. There are two interior functions - the one named **read()** and the other defined in the list being returned named `values`. Note the use

of `<<-` in the **read()** function which modifies the variable `values` shared by both functions, and not just a local copy within the call frame of the **read()** function each time it is invoked.

We can use this code in our call to `getURL` as follows:

```
reader = stockReader()
getURL('http://www.omegahat.org/RCurl/stockExample.dat',
       write = reader$read)
reader$values()
```

We create a specific instance of the **stockReader()**. Each instance has its own separate `values` variable. We pass the **reader()** function in the returned list as the `write` argument of **getURL()**. As libcurl processes the chunks, this functions is called for each chunk and it modifies the `variable` function. At the end of the call to **getURL()**, we can get the final version of `values` by calling the **values()** function of `reader`.

Unfortunately, things are not quite as simple as this. **libcurl**receives the HTTP response in arbitrarily sized chunks. The chunks are not necessarily broken by word or line boundaries. As a result, two consecutive chunks might be split in the middle of a value. And, of course, processing such chunks would lead to errors. Instead, we need to ensure that we process entire words or even lines. The **RCurl** provides a filter function that can be used to do this. The function **chunkToLineReader()** takes the real chunk processor function (e.g. `reader$read` above) and arranges to call it with only complete lines. This allows us to do line-at-a-time processing which is reasonably common for current data formats.

```
reader = stockReader()
getURL('http://www.omegahat.org/RCurl/exampleStock.dat',
       write = chunkToLineReader(reader$read))
mean(reader$values())
```

At this point, `reader$values()` will return all the data.

The same can be done for reading data frames, matrices and other data formats. Similarly, we can parse XML documents using this on-the-fly approach as we will discuss in section 5.2

5.2. Connecting RCurl and XML parsers

This example uses RCurl to download an HTML document and then collect the name of each link within that document. The purpose of the example is to illustrate how we can combine the RCurl package to download a document and use this directly within the XML (or HTML) parser without having the entire content of the document in memory. We do not save the XML document to disk or even to an R variable, but rather arrange for the XML parser to turn to **libcurl** when it needs more content to process. We start the HTTP download and pass a function to the **xmlEventParse()** function for processing. As that XML parser needs more input, it fetches more data from the HTTP response stream. In this way, the XML parser is "pulling" data from the HTTP request in a "just-in-time" fashion. This is useful for handling very large data that is returned from Web queries. If the HTTP request is simple, we can make use of the XML parser's own HTTP facilities and avoid the use of **RCurl**. However, if the request requires any non-trivial HTTP features, we need a more advanced HTTP client.

To do this, we need to use the **multi** interface for **libcurl** in order to have asynchronous or non-blocking downloading of the document. The steps are quite simple. We initiate the download and associate a "writer" to gather the body of the HTTP response. This is registered with **libcurl** via the *write* callback option and is invoked whenever **libcurl** is in control and is processing the HTTP response. If there is information to be read on the HTTP stream from the server, this function reads it and appends it to a variable `pending`.

The second part of this mechanism is that we need a function that is called by **xmlEventParse()** which can provide input to the XML parser when it requires content. Of course, it will use the content coming from the HTTP server that is collected in the function **getHTTPResponse()**. So we create a sibling function of our write callback function that shares the state of the **getHTTPResponse()** function and so can access the current contents of the variable `pending`. When the XML parser demands some input, our function **supplyXMLContent()** checks to see if `pending` has non-trivial content (i.e. is not the empty string). If it has some content, it returns that. Otherwise, it tells **libcurl** to read some more from the HTTP stream. When it hands control to **libcurl** in this way, **libcurl** will invoke our **getHTTPResponse()** function, populating the contents of `pending`. So when **libcurl** yields control, we will now have content to pass to the XML parser.

The only additional issue that we have to deal with in this setup is that the XML event parser asks for input up to a certain size. We cannot necessarily give it all of the content of `pending`. If `pending` has more characters than the XML parser wants, we must give it the first *`maxLen`* characters and then leave the remainder in *`pending`* for the next request from the XML parser.

The following generator function defines the two functions that do the pulling of the text from **libcurl** and the pushing to the XML parser.

```
HTTPReaderXMLParser =
function(curl, verbose = FALSE, save = FALSE)
{
         # currently available content from HTTP request
   pending =  ""
         # if save = TRUE, holds entire document on completion
   text = character()

   getHTTPResponse =
   function(txt) {

      pending <<- paste(pending, txt, sep = "")

      if(save)
        text <<- c(text, txt)

      if(verbose) {
        cat("Getting more information from HTTP response\n")
        print(pending)
      }
```

```
    TRUE
 }

supplyXMLContent =
 function(maxLen) {
    if(verbose)
      cat("Getting data for XML parser\n")


   if(pending == "") {

       if(verbose)
        cat("Need to fetch more data for XML parser from HTTP response\n")

       while(pending == "") {
          status = curlMultiPerform(curl, multiple = TRUE)
          if(status[2] == 0)
             break
       }
   }

   if(pending == "")
       # There is no more input available from this request.
     return(character())


    # Now, we have the text, and we return at most maxLen - 1
    # characters
   if(nchar(pending) >= maxLen) {
      ans = substring(pending, 1, maxLen-1)
      pending <<- substring(pending, maxLen)
   } else {
      ans = pending
      pending <<- ""
   }

   if(verbose)
      cat("Sending '", ans, "' to XML\n", sep = "")

   ans
 }

 list(getHTTPResponse = getHTTPResponse,
      supplyXMLContent = supplyXMLContent,
      pending = function() pending,
      text = function() paste(text, collapse = "")
     )
```

```
}
```

The remaining part involves combining these pieces with **RCurl** and the **XML** packages to do the parsing in this asynchronous, interleaved manner. The code below performs the basic steps

```
handle = getCurlMultiHandle()
streams = HTTPReaderXMLParser(handle)

uri = "http://www.omegahat.org/RDoc/overview.xml"
getURLAsynchronous(uri,
                   write = streams$getHTTPResponse,
                   multiHandle = handle,
                   perform = FALSE)

links = getDocbookLinks()
xmlEventParse(streams$supplyXMLContent, handlers = links, saxVersion = 2)
links$links()
```

The steps in the code are explained as follows. We first create a 'multi handle'. This gives us the asynchronous behavior that returns control back to us from **libcurl** rather than sending the request and slurping back all the data in one single atomic action. Next, we create our functions to do the pulling and pushing of text from HTTP to the XML parser. These are returned from the call to **HTTPReaderXMLParser()**. And we then setup the request to fetch the content of the URI with the call to **getURLAsynchronous()**. Note that we tell it not to actually perform the request, i.e. `perform = FALSE`. We are just setting it up to be done when the XML parser requests input. This is important as this call must return so that we can call **xmlEventParse()**[3]. The next step is to establish the XML event parser. We provide a collection of XML element parsing handlers that process the XML content in the way that we want (see below). And now we are off, and the XML parser will request input and the functions will read from the HTTP stream.

To process the links within the Docbook document, we are looking for each <ulink> element and fetching its url attribute. So we can provide a collection of handlers that consist of a function only for ulink. And it need only look at the attributes it is given and determine if there is a url entry. If there is, it appends the value to its internal collection of links. When we are finished the parsing, we can ask for this collection of links using the additional function links.

```
getDocbookLinks =
function()
{
 links = character()

 ulink = function(name, attrs, ns, namespaces) {
```

[3]If we did perform the request, we would merely start the download and perhaps slurp up some of the response. This would still be available to the XML parser so no data would be lost. It may just marginally spoil the efficiency of the approach, but really only marginally if at all.

```
    if("url" %in% names(attrs))
      links[length(links) + 1 ] <<- attrs["url"]
 }

 list(ulink = ulink,
      links = function() links)
}
```

To run this code, we need to load both the **RCurl** and **XML** packages.

5.3. Multiple Concurrent Downloads using RCurl

In this example, we look at how we can send multiple HTTP requests and process them concurrently. The basic idea is as follows. We specify a collection of URIs to download when establishing the HTTP requests, but don't send any of the requests until all the requests are constructed. Then, we send all of the requests and get ready to harvest the results as they are returned by the different Web servers and connections. As each Web server sends back a piece of the response, we collect that and then return to processing the responses on the other connections. When the last one finishes, we return control to the caller.

This is quite different from processing the requests sequentially, waiting for one to finish before starting the next. The expectation is that this concurrent approach will be faster than the serial version. When we perform the requests sequentially, one slow request will mean our client will essentially be idle waiting to establish a connection to the Web server or waiting for a piece of the response. In our interleaved, concurrent approach, the client can continue to process the other requests while waiting for the server to respond on a slow connection.

This approach is also different from processing all the requests concurrently but in the background. In that case, having dispatched the requests, control would be returned to the caller and R would be able to do other things. Notification of pending content from the request would need to be done via the event loop. This is feasible (at least on Unix), but different from the example we are describing here. Here, the processing of the set of requests is a blocking action, but each request is processed asynchronously.

The implementation requires using the "multi" interface for **libcurl**. We create a multi handle and then we create a regular curl handle for each individual request, i.e. for each URI to be fetched. We add each of these regular/easy curl handles to the multi handler and then call **curlMultiPerform()** until it terminates. Terminating means either an error or that each of the requests has completed.

```
getURIs =
function(uris, ..., multiHandle = getCurlMultiHandle(), .perform = TRUE)
{
  content = list()
  curls = list()

  for(i in uris) {
    curl = getCurlHandle()
    content[[i]] = basicTextGatherer()
    opts = curlOptions(URL = i, writefunction = content[[i]]$update, ...)
```

```
    curlSetOpt(.opts = opts, curl = curl)
    multiHandle = push(multiHandle, curl)
  }

  if(.perform) {
     complete(multiHandle)
     lapply(content, function(x) x$value())
   } else {
     return(list(multiHandle = multiHandle, content = content))
   }
}
```

We will do some very simple timing experiments to see how the different approaches perform. These are intended to give an approximate idea of the relative performances. They should not be taken extremely seriously as there are many different factors for which we need to control such as the network on which the tests were run, the number and domains of the test pages, etc.

We will use 5 documents from quite different domains and geographic locations.

```
uris = c("http://www.omegahat.org/index.html",
         "http://www.r-project.org/src/contrib/PACKAGES.html",
         "http://developer.r-project.org/index.html",
         "http://www.slashdot.org/philosophy.xml",
         "http://fxfeeds.mozilla.org/rss20.xml",
         "http://www.nytimes.com/index.html")
```

To see what effect we have on timing, we run the asynchronous version and the serial version with the commands

```
asyncTimes = replicate(100, system.time(getURIs(uris)))
serialTimes = replicate(100, system.time(getURI(uris, async = FALSE)))
```

We get three measurements for each of the 100 replicates, giving the amount of time consumed within the R process only, within the system-level calls and the total elapsed times.

Figure 1 and table 1 illustrate the difference between the performance of the two methods. It is clear that there is a lot of variability, with some of the downloads taking up to 20 seconds. And it is also clear that about 80% of the times for the asynchronous downloads are smaller than the minimum of the 100 serialized downloads. So there is an improvement in performance, at least in this simple experiment. The distributions of user and system times are remarkably similar for the two different approaches. This is a little surprising as we would expect the asynchronous version would spend more time switching between the downloads, checking to see if there is anything to be read on each connection and so be less idle and thus consuming more user and system time. The serialized version does not have to do this switching but can remain idle until there is input waiting on the particular connection on which it is focusing. We have seen this trade-off between elapsed time and CPU time in other experiments with different versions of the software. It is a convenient one - the user can chose whether their time or the computer's time is more important.

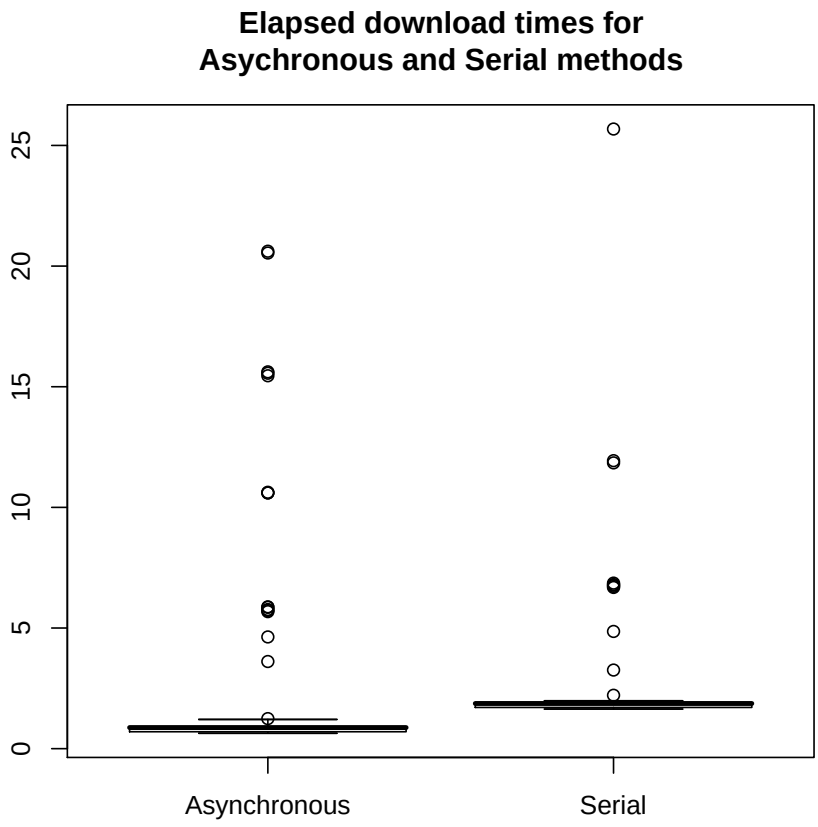

Figure 1: Comparison of the download times for the concurrent and serialized requests of the 5 documents.

	Minimum	1^{st} Quartile	Median	3^{rd} Quartile	Maximum
Concurrent	0.638	0.701	0.875	0.909	20.600
Serial	1.64	1.71	1.87	1.91	25.70

Table 1: Distribution of elapsed download times for 100 replicates of the concurrent and serial HTTP requests of 5 documents.

5.4. Nested HTML Downloads

The goal of this example is to show how we can parse an HTML document and download each of the files to which it has links, e.g <a href=...> elements. This is a simple Web crawler or robot. As we process the original document, we arrange to download the others. There are three possible "obvious" approaches.

- One approach is to parse the original document in entirety and extract its links (either by fetching the document and then parsing it or "on the fly" by connecting the output of the curl request directly to the xml parser). We then download each of those linked documents.
- Another approach is to start the parsing of the top-level document and when we encounter a link, we immediately recursively download that document and then continue on with the parsing of the original document. In other words, when we encounter a link, we hand control to the downloading of that link.
- An intermediate approach is to parse the first document and as we encounter a link, send a request to download that document and arrange to have it be processed concurrently with the other documents. Essentially, we arrange for the processing of the links to be done asynchronously. Having encountered a link, we don't wait until it is completely downloaded and nor do we wait to download all of the links after we have processed the original document. Rather, we add a request to download the link as we encounter it and continue processing.

The strategy in this approach is to start the parsing of the original document. We do this in almost exactly the same way that we do in the XML parsing example above (see 5.2). That is, we create a multi CURL handler and we provide a function that will feed data from the HTTP response to the XML parser when it is required. We then put the downloading of the original/top-level file on the stack for the multi handler

```
uri = "http://www.omegahat.org/RCurl/philosophy.xml"

multiHandle = getCurlMultiHandle()
streams = HTTPReaderXMLParser(multiHandle, save = TRUE)

curl = getCurlHandle(URL = uri, writefunction = streams$getHTTPResponse)
multiHandle = push(multiHandle, curl)
```

At this point, the initial HTTP request has not actually been performed and therefore there is no data. And this is good. We want to start the XML parser. So we establish the handlers that will process the XML elements of interest in our document, e.g. a <ulink> for a Docbook document, or <a> for an HTML document. The function **downloadLinks()** defined below is the function used to do this. And now we are ready to start the XML parser via a call to **xmlEventParse()** in the **XML** package.

```
links = downloadLinks(multiHandle, "http://www.omegahat.org",
                      "ulink", "url")
xmlEventParse(streams$supplyXMLContent, handlers = links, saxVersion = 2)
```

At this point, the XML parser asks for some input. It calls the **supplyXMLContent()** and this fetches data from the HTTP reply. In our case, this will cause the HTTP request to be sent to the server and we will wait until we get the first part of the document. The XML parser then takes this chunk and parses it. When it encounters an element of interest, e.g. a <ulink>, it calls the approriate handler function given in `links`. And this gets the URI of the link and then arranges to add to the multi handle an HTTP request to fetch that document. The next time that the multi curl handle is requested to get input for the XML parser, it will send that new HTTP request and the response will be available. The *writefunction* handler for the new HTTP request simply collects all the text for the document into a single string. We use **basicTextGatherer()** for this.

There is one last little detail before we can access the results. It is possible that the XML event parser will have digested all its input before the downloads for the other documents have finished. There will be nothing causing **libcurl** to return to process those HTTP responses. So they may be stuck in limbo, with input pending but nobody paying attention. To ensure that this doesn't happen, we can use the **complete()** function to complete all the pending transactions on the multi handle.

```
complete(multiHandle)
```

And now that we have guaranteed that all the processing is done (or an error has occurred), we can access the results. The result of calling **downloadLinks()** gives us a function to access the download documents:

```
links$contents()
```

To get the original document in addition to its links, we have to look inside the `streams` object and ask it for the contents that it downloaded. This is why we called **HTMLReaderXMLParser()** with `TRUE` for the *`save`* argument.

The definition of the XML event handlers is reasonably straightforward at this point. We need a handler function for the link element that adds an HTTP request for the link document to the multi curl handle. And we need a way to get the resulting text back when the request is completed. We maintain a list of text gatherer objects in the variable `docs`. These are indexed by the names of the documents being downloaded.

The function that processes a link element in the XML document merely determines whether the document is already being downloaded (to avoid duplicating the work) or not. If not, it pushes the new request for that document onto the curl handle and returns. This is the function **op()**.

There are details about dealing with relative links. We have ignored them here and only dealt with links that have an explicit `http` prefix. The function that is used to make the requests and store the results is then defined as follows:

```
downloadLinks =
function(curlm, base, elementName = "a", attr = "href", verbose = FALSE)
{
   # stores the reader function for each document
 docs = list()
```

```
 # accessor for the result on completion, returning all of the documents
contents = function()
   sapply(docs, function(x) x$value())


ans =  list(docs = function() docs,
            contents = contents)

  # Process the XML node and get the
op = function(name, attrs, ns, namespaces) {

  if(attr %in% names(attrs)) {

     u = attrs[attr]
     if(length(grep("^http:", u)) == 0)
        return(FALSE)

     if(!(u %in% names(docs))) {
        if(verbose)
           cat("Adding", u, "to document list\n")
        write = basicTextGatherer()
        curl = getCurlHandle(URL = u, writefunction = write$update)
        curlm <<- push(curlm, curl)

        docs[[u]] <<- write
     }
  }

  TRUE
}

 # Use op as the XML parsing handler for each XML element name of interest
ans[elementName] = op

ans
}
```

6. Alternative Approaches and Related and Future Work

We mentioned previously that we can achieve basic, high-level support for HTTP directly in R using the socket connection mechanism or via the HTTP and FTP code in R ported from the libxml source. One advantage of these approaches is, of course, that R users do not need to install any additional software to access these functions. Packages that use HTTP requests via this mechanism do not have any additional dependencies. To make a request via **RCurl** on the other hand, a user must install both **RCurl** and libcurl itself. The installation of the R package is almost transparent given the good tools that are built into R for distributed

package updating. libcurl is also relatively easy to install. Binaries are distributed for a large number of platforms.

R also provides built-in functions such as **download.file()** and **url()**. The former copies the contents of the target URI to the file system. Unfortunately, to read this information back into R therefore requires two passes of the contents of the file - one for the download, and one for the loading into R. Using **url()**, we can avoid a unnecessay second pass of the data as this function allows us to read the contents directly from the Web server into R. And we can use the socket connection tools in R to have even lower-level access to the infrastructure on which we can build an HTTP client. Naturally, this general communication mechanism does not have any knowledge of the HTTP format. So we would have to provide higher-level functionality such as escaping characters, secure connections, support for cookies, etc.

The R package **httpRequest** available from CRAN provides a higher-level interface to the HTTP requests using R's basic socket facilities. The package is intentionally simple, providing the essentials for downloading URIs, and submitting forms and multi-part POST requests. As the documentation explains, it does not attempt to escape characters; that is left for the caller. Also, it does not handle chunked responses. And it intentionally does not attempt to provide a general interface for adding fields to the HTTP request header other than the referer field, or handling cookies, SSL connections, etc. Our original work in this area explored the same approach and built some additional features in a "pure" R package. However, the details of the various features quickly become overwhelming, eventhough we are working in a high-level language. We abandoned that work in favor of interfacing to a library that has ongoing development and wide use.

Having made the decision to use a third-party library to provide the HTTP client facilities in R, there were still decisions to be made about which approach to take. It is possible to use one of the inter-system interfaces such as the **RSPerl** Temple Lang (2006a), **SJava** Temple Lang (2005) or **RSPython** Temple Lang (2006b) packages and use the existing facilities of that other language to act as a Web client. Alternatively, the more traditional approach, and therefore potentially simpler to socialize with users, is to use a C/C++ library that provides the relevant services. And there are many potential candidates. A reasonably comprehensive list of serious possibilities is given at `http://curl.haxx.se/libcurl/competitors.html`. Each has different features, different levels of suppport and active development, and are more or less portable. The developers of libcurl do appear to have a keen focus on portability and that is a significant advantage. Other libraries such as libwww, libghttp, libferit, neon, libsoup, mozilla netlib, http-tiny, fetch all have their own merits. Neon's support for Distributed Authoring and Versioning (WebDAV) extensions to HTTP make it interesting. In the end, libcurl's extensive facilities, relatively simple programming interface (API) and extensive portability makes it at least a good choice.

6.1. Future Work

RCurl is reasonably full-functioned as a general HTTP client. Of course, we could continue to provide higher-level utility functions that would further simplify its use for common situations. Our focus has instead been on providing a general and flexible infrastructure on which such high-level functions can be constructed by those working entirely in R.

While the infrastructure is relative complete, there are some additional elements that would provide useful functionality.

Uploading Files, Binary Content and Connections HTML forms provide a way to specify the name of a file whose contents are to be sent as part of a form submission. The function **postForm()** could be made to assist users in this endeavor rather than having them read the contents of the file and pass this to **postForm()**. The function could recognize connection objects or file names in R as special types of arguments and transfer the contents to the body of the HTTP request. The readfunction option for libcurl requests can be used to do this in different, general ways. There is a complicating factor since the contents may be in a binary format rather than ASCII and so harder to deal with directly via text manipulation in R.

Indeed, we often download and upload binary content from and to a Web server. For example, we might download an image file in JPEG or PNG format, or we might upload an Rda file containing serialized R data to a web site. At present, the user is on her own in doing this with the **RCurl** package. Conceptually it is not very hard to add support for this. When downloading binary data, the user can specify a C routine that would take over the processing of the body of the response. Alternatively, when passing the data to a user-supplied writer function, we can provide it as raw bytes. For uploading binary data in a request, we can use the same approaches.

However, it would be generally useful to make HTTP requests, both the outgoing request and the incoming response behave more like connections. Unfortunately, the connection C-level API in R is not a published one. It is problematic for packages to define new connection types or extend existing ones.

Classes for URIs It would be useful to have explicit formal (S4) classes to represent URIs. The natural representation of protocol (e.g. http), port (e.g. 80 for HTTP), domain (www.omegahat.org) and file path (/RCurl/index.html) would facilitate processing such inputs. For example, when processing relative URIs such as `<a href=...>` elements in HTML and `<ulink url=...>`, we need to be able to easily compute the fully-qualified URI of the relative link by merging it with the base URI. These classes are not necessarily best located within the **RCurl** package as they are applicable in wider contexts. Rather, they probably belong in base R and may be added in the future. At present, there are some facilities for this in the XML package (see http://www.omegahat.org/RSXML).

Exceptions We have a great deal of information from the **libcurl** library about the nature of errors that arise during the course of an HTTP request. As with many R packages, we should make use of an explicit class hierarchy of exception types so that programmers using this package can provide error handlers for specific types of errors.

Event Loop We have used **libcurl** to provide asynchronous HTTP requests. In this context, our application is telling **libcurl** when it needs more data from these requests. It is also common to want to initiate asynchronous requests and to have **libcurl** inform the application when data is available. This is how most Web browsers work and how we would like to be able display information within a graphical interface for R. It is possible for us to access information from **libcurl** that allows us to integrate the request response into the R event loop, at least on UNIX platforms. This can be done using the the routines already in R for registering external sources of events via file descriptors, or using the **REventLoop** package (see http://www.omegahat.org/REventLoop).

SSL Support We will increasingly access data dynamically from Web servers as regular documents or via Web service methods and confidentiality and security will become more essential. As we make further strides in secure communication within statistical computation, we will need access to the features of **libcurl** that allow us to control the behavior of the SSL engine. This includes how certificates are found and authenticated. While **libcurl** provides simple-to-use options for controlling some of this, we need to simplify the ways in we can customize the creation of the secure connections. Indeed, it would be beneficial to have more general support for SSL in R via a separate package.

Automated Code Generation The interface to **libcurl** includes information mapping the individual options, errors, etc. to enumerated values in the C code. We have constructed these by hand from specific versions of **libcurl**. As **libcurl** evolves, new options and values may be introduced, others removed. These changes will not be included in our interface unless we update it regularly. Using some tools we are developing separately from this package – the package **RGCCTranslationUnit** – we can process the **libcurl** source code and generate the interface code automatically.

7. Summary

We outlined the basic structure of HTTP requests that underlie so much of the facilities provided on the Web. We described an R package - **RCurl** - that provides high-level facilities for R programmers to make HTTP requests to servers using a C-level library, **libcurl**. The basic functionality allows one to download documents and submit forms. The architecture of the package allows for a great deal of control by the programmer for more complex queries. It includes an interface to controlling authentication, secure connections and more by leveraging those facilities provided by **libcurl**. The package allows for sophisticated and efficient asynchronous request processing. This package already provides the backbone of the client-side Web services facility in the **SSOAP** Temple Lang (2006c) package, and the **HTMLForms** package. It forms a rich starting point on which other packages using HTTP can be built.

References

(2004). "Sprint IPMON DMS - Application Breakdown." http://ipmon.sprintlabs.com/packstat/viewresult.php?0:appsbreakdown:sj-20.0-040206.

Chambers JM (1999). *Programming with Data*. Springer Verlag.

Claffy K, Miller G (1998). "The Nature of the Beast: Recent Traffic Measurements from an Internet Backbone." In "INET '98," Internet Society.

Fielding R, Gettys J, Mogul J, Frystyk H, Masinter L, P Leach TBL (1999). "Hypertext Transfer Protocol - HTTP/1.1." http://www.w3.org/Protocols/rfc2616/rfc2616.html.

Gentleman R, Ihaka R (200). "Lexical Scope and Statistical Computing." *Journal of Computational and Graphical Statistics*, **9**, 491–508.

R Development Core Team (2005). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.

Snell J, Tidwell D, Kulchenko P (2002). *Programming Web Services with SOAP*. O'Reilly.

Stenberg D, the cURL development team (2006). "libcurl curl easy setopt documentation." http://curl.haxx.se/libcurl/c/curl_easy_setopt.html.

Temple Lang D (2005). "The SJava package for R." http://www.omegahat.org/SJava.

Temple Lang D (2006a). "The RSPerl package for R." http://www.omegahat.org/RSPerl.

Temple Lang D (2006b). "The RSPython package for R." http://www.omegahat.org/RSPython.

Temple Lang D (2006c). "The SSOAP package for R." http://www.omegahat.org/SSOAP.

Temple Lang D (2006d). "The XML package for R." http://www.omegahat.org/RSXML.

Veillard D (2006). "libxml: The XML C parser and toolkit of Gnome parsing." http://www.xmlsoft.org.

Viega J, Messier M, Chandra P (2002). *Network Security with OpenSSL*. O'Reilly Media.

(W3C) WWWC (2006). "HyperText Markup Language (HTML) Home Page." http://www.w3.org/MarkUp/.

8. Glossary

We provide a short, simple description of some of the acronyms and terms used in the paper. More information can be found on the Web, e.g. http://www.wikipedia.org.

HTTP HyperText Transfer Protocol

SOAP Simple Object Access Protocol

URI Uniform Resource Identifier. URL was used to signify a Uniform Resource Locator.

Cookie A "cookie" is a small piece of information sent by a Web server to be stored by a web browser so it can later be read back in subsequent requests from that browser. This is useful for having the browser remember some specific information about that browser/user.

SSL or TLS Secure Socket Layer. This is mechanism for securing the communication over a low-level socket connection between two machines so that an other application cannot intercept and interpret the data in a meaningful way. The information is encrypted.

TFTP trivial file transfer protocol. This is a very simple form of file transfer which can be implemented relatively easily and uses a very small amount of memory. It provides no security mechanisms. This is used, for example, for booting machines over a network for thin-clients and routers.

GSS Generic Security Service.

NTLM Windows NT LAN Manager is an authentication protocol used in various Microsoft network protocol implementations and supported by the NTLM Security Support Provider ("NTLMSSP"). Originally used for authentication and negotiation of secure Remote Procedure Calls (RPC), NTLM is also used throughout Microsoft's systems as an integrated single sign-on mechanism. See http://curl.haxx.se/rfc/ntlm.html for details.

CRAN Central R Archive Network which is a repository for a large collection of R packages and related software.

Affiliation:

Duncan Temple Lang
Department of Statistics,
371 Kerr Hall,
One Shields Avenue,
Davis
CA 95616
U.S.A. E-mail: duncan@r-project.org
URL: http://www.stat.ucdavis.edu/~duncan, http://www.omeghat.org

Journal of Statistical Software *Submitted:* yyyy-mm-dd
MMMMMM YYYY, Volume VV, Issue II. *Accepted:* yyyy-mm-dd
http://www.jstatsoft.org/